VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

BEZPEČNÉ NALEZENÍ ZDROJŮ REST ARCHITEKTURY NA ZÁKLADĚ JEJICH SÉMANTICKÉHO POPISU

SECURE SEMANTICALLY-BASED DISCOVERY OF RESOURCES IN RESTFUL ARCHITECTURE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JIŘÍ KOUDELKA**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/18633/2017/xkoude19

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Koudelka Jiří, Bc.**
Obor: Informační systémy
Téma: **Bezpečné nalezení zdrojů REST architektury na základě jejich sémantického popisu**
Secure Semantically-Based Discovery of Resources in RESTful Architecutre
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s architektonickým stylem REST a s reprezentací REST zdrojů ve webových službách. Prozkoumejte způsob hledání poskytovatelů webových služeb a zdrojů pomocí protokolů WS-Discovery pro SOAP a mRDP pro REST webové služby. Podrobněji se věnujte protokolu mRDP vč. možností sémantického popisu služeb.
2. Navrhněte vlastní řešení či rozšíření mRDP protokolu pro bezpečné nalezení REST zdrojů - konkrétně řešte autentizaci poskytovatelů zdrojů, autorizaci klientů a důvěrnost a integritu komunikace. Diskutujte možné případy užití.
3. Po konzultaci s vedoucím implementujte navržené řešení pro bezpečné nalezení zdrojů REST architektury na základě jejich sémantického popisu jako Java knihovnu použitelnou v aplikacích OS Android. Implementujte také ukázkovou aplikaci, tj. poskytovatele zdrojů a klienty k nim přistupující.
4. Výsledek otestujte, zveřejněte jako open-source a navrhněte možná rozšíření.

Literatura:

- Vazquez, J.I.; López-de-Ipia, D., "mRDP: An HTTP-based lightweight semantic discovery protocol", Computer Networks, 51(16), pp. 4529-4542, 2007. ISSN 1389-1286. [http://dx.doi.org/10.1016/j.comnet.2007.06.017].
- Moritz, G.; Zeeb, E.; Pruter, S.; Golatowski, F.; Timmermann, D.; Stoll, R., "Devices Profile for Web Services and the REST". In 8th IEEE International Conference on Industrial Informatics (INDIN), pp. 584-591, IEEE, Osaka, 2010. ISBN 978-1-4244-7298-7. [http://dx.doi.org/10.1109/INDIN.2010.5549678]
- Sepulveda, C.; Alarcon, R.; Bellido, J., "QoS aware descriptions for RESTful service composition: security domain". World Wide Web, 18(4), pp. 767-794, 2015. ISSN 1573-1413. [http://link.springer.com/article/10.1007/s11280-014-0278-0]

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 a započatá práce na řešení bodu 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Diplomová práce se zabývá problematikou bezpečného nalezení zdrojů REST architektury na základě jejich sémantického popisu. Předmětem práce je implementace a případné rozšíření protokolu mRDP pro bezpečné nalezení REST zdrojů, včetně implementace odpovídající open-source knihovny kompatibilní s operačním systémem Android. Další oblastí práce je implementace jednoduchých ukázkových aplikací, které tuto knihovnu využívají.

Abstract

This thesis looks into the problematics of secure semantically-based discovery of resources in REST-ful architecture. The subject of this thesis is the implementation and potential extension of the mRDP for secure discovery of resources in REST-ful architecture, including the implementation of a corresponding open-source library compatible with the Android operating system. Another topic of this thesis is the implementation of simple example applications using this library.

Klíčová slova

REST, mRDP, Android, URL, SPARQL, Plant, HTTP, HTTPS, OpenmRDP.

Keywords

REST, mRDP, Android, URL, SPARQL, Plant, HTTP, HTTPS, OpenmRDP.

Citace

KOUDELKA, Jiří. *Bezpečné nalezení zdrojů REST architektury na základě jejich sémantického popisu.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Bezpečné nalezení zdrojů REST architektury na základě jejich sémantického popisu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Marka Rychlého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Koudelka
22. května 2018

Poděkování

Děkuji svému vedoucímu diplomové práce RNDr. Marku Rychlému, Ph.D. za odborné vedení a rady, které mi poskytl a pomohl tak s řešením práce. Dále děkuji panu Dr. Iñaki Vázquez za rady a infromace týkající se protokolu mRDP.

Obsah

Kapitola 1

Úvod

V současné době je jednou z nejvíce využívaných architektur pro webové služby architektura REST (Representational State Transfer). Narozdíl od dalšího často využívaného protokolu SOAP, však REST ve své originální definici, nedisponuje možností nalezení zdrojů. Nalezení zdrojů bez znalosti přesného umístění je obzvláště důležité v době mobilních zařízení, kdy se neustále mění poloha (a tedy i adresa) zdroje, který je na mobilním zařízení. Dalším příkladem využití pak může být například dohledání zdroje, který splňuje některé konkrétní podmínky nebo zdroje, o kterém víme pouze jak se jmenuje, ale neznáme jeho přesné umístění.

Tato diplomová práce se zabývá problematikou bezpečného nalezení zdrojů REST architektury na základě jejich sémantického popisu. Jejím předmětem je seznámení se s již existujícími způsoby sémantického výhledávání (především pak s teoretickým návrhem protokolu mRDP [10] (multicast Resource Discovery Protocol)), návrh možných rozšíření pro tento teoretický protokol a samotná implementace odpovídající open-source knihovny v jazyce Java s podporou pro operační systém Android.

Druhá kapitola je věnována stručné analýze již existujících řešení, především pak teoretickému návrhu protokolu mRDP, na kterém bude výsledná knihovna postavena. Dále se pak tato kapitola zaměřuje na návrh implementace vlastní Java knihovny, vývojového a testovacího prostředí a testování výsledné implementace.

Ke správnému pochopení problematiky sémantického vyhledávání v architektuře REST se ve třetí kapitole zabývá práce základním popisem této architektury, zdroji se kterými REST pracuje, adresací těchto zdrojů a způsobem reprezentace zdrojů.

Ve čtvrté kapitole následuje seznámení s obecným problémem sémantického vyhledávání. Je zde představen samotný koncept sémantického vyhledávání a prostředky, které jsou pro uskutečnění tohoto vyhledávání využívány, jako například Resource Definition Framework nebo Web Ontology Language. Dále je v této kapitole uveden i názorný příklad, jak takové sémantické vyhledávání zdrojů probíhá.

Pátá kapitola je věnována teoretickému návrhu protokolu mRDP, ze kterého do značné míry vychází celá implementace výsledné knihovny. Je zde představen základní koncept tohoto protokolu, návrh zabezpečení komunikace, příklad samotné komunikace mezi mRDP klientem a mRDP servery a ukázka formátu zpráv, pomocí kterých komunikují mRDP klient a mRDP server.

Následující šestá kapitola je zaměřena na vlastní implementaci samotné knihovny. Jsou zde popsány jednotlivé balíky knihovny OpenmRDP a vyjímky, naprogramovány za účelem ošetření chybových stavů. Dále je zde podrobně představena implementace objektu reprezentující základní zprávu a implementace jednotlivých typů zpráv včetně jejich serializace

a deserializace. Další podkapitolou je zde představení implementace funkcionality `LOCATE` spolu s představením zpracování báze znalostí, ontologie a manažera informací. V této kapitole se poté nachází i představení implementace druhé funkce, kterou knihovna OpenmRDP nabízí, a to funkce `IDENTIFY` včetně podpůrných tříd. Velmi důležitou součástí této kapitoly je také popis veřejných rozhraní, které knihovna poskytuje programátorům, představení obsluhy HTTP požadavků, podpůrných objektů a implementace zaznamenávání událostí.

Po kapitole, která představuje vlastní implementaci, je zde kapitola sedmá. Tato kapitola se zaměřuje na představení konceptu testování knihovny. Jsou zde popsány automatizované testy, které doprovázely celý proces implementace knihovny, testovací prostředí, kterým se pro knihovnu OpenmRDP rozumí aplikace testovacího klienta, aplikace testovacího serveru a aplikace klienta pro mobilní operační systém Android.

Poslední osmá kapitola se věnuje využití knihovny OpenmRDP v praxi. Jsou zde diskutovány návrhy jejího využití v budoucnosti, ukázka síťové komunikace při jejím využití a na závěr její dostupnost.

Kapitola 2

Analýza a návrh řešení

Knihovna pro podporu bezpečného nalezení zdrojů REST architektury, na základě jejich sémantického popisu, byla navržena pro platformu Java (Knihovna by měla být kompatibilní s operačním systémem Android) a měla by být po její implementaci distribuována pod licencí open-source. Dle zadání se má jednat o knihovnu, která uživateli umožní identifikovat zdroj a nalézt jeho lokaci na základě vstupních podmínek nebo pomocí jeho pojmenování. Tedy bez znalosti jeho přesného umístění. K nalezení tohoto zdroje by měly uživateli stačit pouze popisné (sémantické) informace o hledaném zdroji. Celé nalezení dat by mělo být bezpečné. Důraz je zde kladen především na autentizaci poskytovatelů zdrojů, autorizaci klientů a důvěrnost a integritu komunikace. Knihovna by měla být pouze rozšiřující a její nasazení na server, poskytující již existující REST API, by nemělo nijak ovlivnit stávající funkcionalitu.

2.1 Existující návrhy vyhledávacích protokolů

V minulosti se využívaly různé vyhledávací protokoly, ať už to byl SLP, Jini nebo například UPnP SSDP. Tyto protokoly však narážely na své vlastní limitace, které byly objeveny [4]:

- **Nedostatečné zastoupení** – žádný z protokolů nebyl tak rozšířený, aby stávající architektura poskytla kompletní podporu pro jeho programovací jazyk nebo nástroje, které využíval.
- **Nedostatek sémantického přizpůsobení** – většina protokolů vyžaduje přesnou sémantickou shodu a není schopna řešit obecný popis zdroje.

2.2 Návrh řešení s využitím protokolu mRDP

Protokol mRDP, tak jak ho navrhli doktoři Vazquez a López-de-Ipiña [10], je po teoretické stránce pro účel sémantického vyhledávání s využitím achitektury REST dostatečný a bude naimplementován podle jejich teoretické předlohy. Komunikace bude probíhat za použití jak protokolu UPD tak TCP.

Knihovna bude mít dva logické celky. Jeden se bude zabývat podporou cílené funkcionality na straně klienta a druhý na straně serveru. Oba celky budou platformně nezávislé s jedním omezením. Jak klient, tak i server musí mít běhové prostředí, které umožní spouštět program v jazyce Java.

UDP protokol bude využit pro multicastové dotazy na hledané zdroje, které budou posílány prostřednictvím zpráv ve formátu mRDP. Odpovědi od serverů, které budou podporovat protokol mRDP, už budou posílány unicastově s využitím protokolů HTTP a TCP.

Servery budou pro vyhodnocování sémantických dotazů používat dva jazyky. Prvním jazykem je jazyk SPARQL, který je pro účel samotného vyhodnocování sémantických dotazů jedním z nejvhodnějších. Nicméně problém nastává s jeho výpočetní náročností [7]. Pokud bychom chtěli server, který bude řešit sémantické dotazy, mít na nějakém méně výkonném zařízení, může být použití tohoto jazyka problémem. Druhý jazyk, který se dá pro vyhodnocování použít, je jazyk Plant (*Pattern Language for N-Triples*), který byl navržen doktorem Vazquezem v jeho disertační práci [11]. Tento jazyk sice není tak výkonný co se kvality vyhledávání týče, ale je pomocí něj možné vyhodnocovat dotazy i na výkonnostně slabších zařízeních. Nově vzniklá knihovna by měla podporovat oba tyto jazyky.

Zabezpečení celé komunikace bude zajištěno za použití mechanismů poskytovaných protokolem HTTP popřípadě HTTPS. Konkrétně tedy autorizace klientů by měla být zajištěna pomocí mechanismu HTTP Digest access authentication. Autentizace poskytovatelů zdrojů, důvěrnost a integrita celé komunikace by potom měla být docílena za pomoci standardů protokolu HTTPS.

2.3 Vývojové a testovací prostředí

Pro vývoj knihovny bude potřeba prostředí, ve kterém se dá již implementovaná funkcionalita testovat. Pro tento účel vznikne topologie, která reflektuje dnešní bežné potřeby. Konkrétně se bude skládat z následujících prvků:

- **Server A** – server běžící na linuxové platformě, disponující RESTovým rozhraním
- **Server B** – server běžící na platformě Windows, disponující RESTovým rozhraním
- **Mobilní telefon** – mobilní telefon s operačním systémem Android, využívající aplikaci, která implementuje cílenou knihovnu
- **Stolní počítač** – klasický stolní počítač s operačním systémem Windows

2.4 Návrh implementace

Implementace knihovny bude uskutečněna v programovacím jazyce Java v aktuální verzi 8 s ohledem na kompatibilitu s operačním systémem Android. Pro vyhodnocování sémantických dotazů budou, jak již bylo zmíněno, využity jazyk SPARQL a Plant.

Pro potřeby testování funkcionality během samotného vývoje a testování bude naimplementována jednoduchá Android aplikace pro mobilní telefon, který bude figurovat jako testovací klient. Dále je třeba nakonfigurovat REST API pro dva testovací servery.

2.5 Návrh testování

Za účelem testování knihovny během vývoje, i po jejím dokončení, bylo navrženo několik způsobů testování. Pro testování aplikace během jejího vývoje bylo naplánováno testování pomocí automatických JUnit testů. Toto testování by mělo být prováděno pravidelně, vždy po implementaci či změně větší části knihovny.

Testování knihovny po jejím dokončení bude prováděno pomocí předem definovaných případů užití a za pomoci předem připraveného testovacího prostředí.

Kapitola 3

Architektura REST

Architektura REST byla představena v roce 2000 v disertační práci Roye Fieldinga. Dříve než se tato architektura začala ve světě informačních technologií používat, byla většina webových služeb dostupná pomocí protokolu SOAP. Tento prokol má i v současné době velké zastoupení, nicméně nelze se ubránit nastupujícímu trendu využívání architektury REST. O tom, že je tato architektura důležitá a hojně využívaná, svědčí například i její využívání velkými technologickými společnostmi:

- Google (Google Fit REST API)
- Facebook (Instagram Graph API)
- Twitter
- Yahoo (Yahoo Social REST APIs)
- Amazon (Amazon S3 REST API)

3.1 Výhody architektury REST

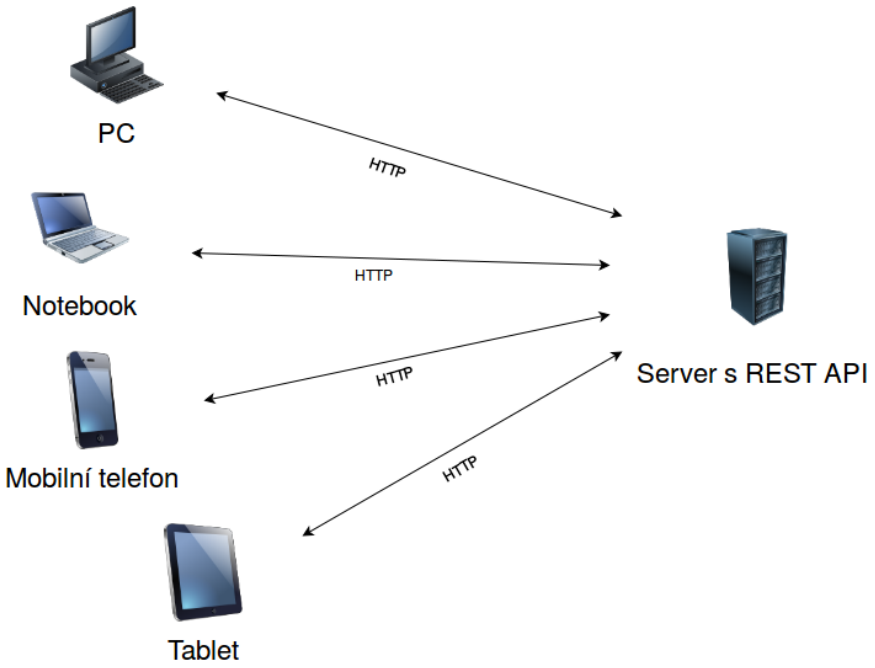
Stále více a více společností přestává používat své zastaralé řešení využívající protokolu SAOP a implemenují nové řešení ve prospěch architektury REST. K tomuto rozhodnutí do jisté míry přispívají i následjící výhody tohoto řešení [3]:

- **Jednoduché vytvoření a údržba** – Jednoduché vytvoření, nastavení a tvorba dokumentace včetně následné údržby.
- **Výkonnost** – Oproti SOAP je jednodušší cachování dat. V kombinaci s menšími zprávami, které se posílají (REST využívá zprávy ve formátu JSON narozdíl od SOAP, který využívá XML), dostáváme rychlejší přístup, který využívá menší šířku přenosového pásma. Tuto vlastnost oceníme převážně u mobilních aplikací.
- **Škálovatelnost** – REST je extrémně flexibilní a dokáže se jednoduše adaptovat na požadavky k rozšíření stávající služby a reflektovat její změny.
- **Různé formáty výstupních dat** – Narozdíl od SOAP nejsou výstupní data omezena pouze formátem XML, ale mohou být v celé množině formátů (Například JSON, CSV, XML, atd.).

- **Kompatibilita s Java Scriptem** – V současnoti je trendem využívat při programování programovací jazyk Java Script. Architektura REST je kompatibilní s tímto programovacím jazykem a dovoluje vytvářet dynamické API s využitím různých nástrojů jako například Node.js.

3.2 Základní princip architektury REST

Jednou z hlavních myšlenek, o které se tato architektura opírá, je univerzálnost. Pomocí REST architektury chceme definovat API, které bude dostupné co největší škále zařízení bez závislosti na operačním systému, ať už se jedná o klasický desktopový počítač, server či mobilní telefon. Proto je důležité, aby tato architektura využívala stávající a dobře známé technologie. V případě REST architektury se pak jedná o protokol HTTP, který se při komunikaci využívá. Celá komunikace je postavena na modelu klient - server a je znázorněna na obázku 3.1.



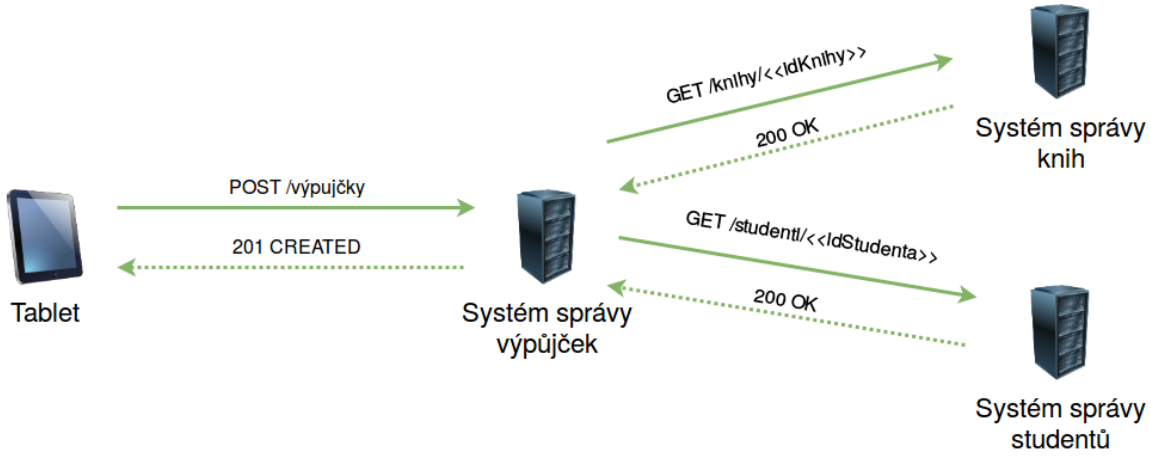
Obrázek 3.1: Komunikace klientů se serverem poskytujícím REST API

3.2.1 REST a práce se zdroji

Další velmi důležitou vlastností architektury REST je její práce se zdroji. Tato architektura totiž narozdíl od staršího přístupu SOAP nevyužívá procedury, ale manipuluje přímo a pouze s daty (zdroji). Zdroj si v praxi můžeme představit například jako objekt typu Student nebo Kniha a samotnou prací s tímto zdrojem může být například požadavek na založení nového studenta v systému, jeho úprava a nebo jen prosté získání informací o studentovi. K této manipulaci se zdroji, jak již bylo zmíněno dříve, se využívá protokol HTTP. Konkrétně by pak tedy naše ukázka se studenty vypadala následovně:

- **GET request** – získání informací o studentovi
- **POST request** – vytvoření nového studenta
- **PUT request** – upravení již existujícího studenta

Celou ukázku toho, jak by mohlo vypadat například zapůjčení knihy studentovi ve školní knihovně, si ukážeme na následujícím obrázku 3.2. Student ze svého tabletu odešle výpůjčnímu systému, který disponuje RESTovým rozhraním, **POST** request na zdroj */výpujčky*. Systém výpůjček odešle **GET** request systému pro správu studentů a systému pro správu knih pro získání potřebných informací (oba tyto systémy taktéž disponují RESTovým rozhraním). Po obdržení potřebných informací, vytvoří systém výpůjček novou výpůjčku a odešle na tablet, který původně posílal **POST** request, odpověď **201 Created**.



Obrázek 3.2: Vytvoření nové výpujčky v systému pro správu výpůjček

3.2.2 Adresace v architektuře REST

Z obrázku 3.2 je patrné, že je třeba znát jejich lokaci nebo jméno, abychom mohli se zdroji pracovat. K popisu takového umístění se využívá **URI** (*Unified Resource Identifier*), a to konkrétně ve dvou svých modifikacích [6].

První a známější modifikací je **URL** (*Unified Resource Locator*) [9]. Jedná se o syntaxi a sémantiku pro kompaktní řetězcovou reprezentaci zdroje dostupného prostřednictvím Internetu. Struktura tohoto protokolu je sekvence znaků (písmena, číslice a speciální znaky), která je zapsána ve tvaru **<schéma>:<schématu-specifická-část>**. Protože, jak již bylo dříve zmíněno, architektura REST využívá protokolu HTTP, je pro nás část **<schéma>** vždy uvažována jako řetězec *http://*.

```
http://example.com
```

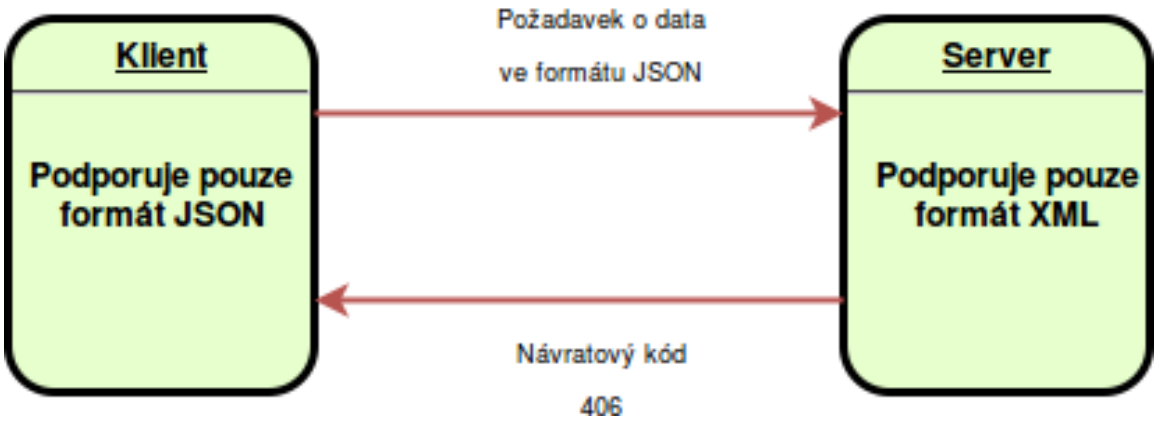
Druhou možností je potom URN (Universal Resource Identifier) [2]. Narozdíl od URL, které nám popisuje, kde se zdroj nachází, URN popisuje, co zdroj ve skutečnosti je. Výhodou URN je nezávislost na lokaci zdroje, tedy není třeba jej měnit, pokud zdroj změní svoje umístění. Každá URN adresa se skládá ze tří částí. První částí je *urn* prefix, dále následuje

namespace identifikátor a nakonec je zde identifikátor zdroje, který je unikátní pro daný *namespace*. Všechny tři části jsou pak oddělené znakem **":"**.

```
URN:ISBN:0-201-67487-4
```

3.2.3 Reprezentace dat

Odpověď RESTového API je v dnešní době nejčastěji reprezentována dvěma formáty (XML a JSON). Nicméně, jak již bylo dříve zmíněno ve výhodách této architektury, není to podmínkou. O tom, v jakém formátu bude odpověď reprezentována, rozhoduje server, který na základě požadavku od klienta dokáže z *Accept* hlavičky požadavku idenfikovat, které formáty jsou pro klienta akceptovatelné. Může ovšem i nastat situace, kdy server není schopen vystavit klientovi odpověď v podporovaném formátu. V takovém případě server odpoví s chybovým stavovým kódem (viz Obrázek 3.3).



Obrázek 3.3: Příklad nekompatibilních formátů (převzato z [6])

V následující ukázce odpovědí budeme uvažovat situaci, kdy si náš výpůjčkový systém z obrázku 3.2 požádal o informace o konkrétním studentovi. Tento požadavek by tedy mohl vypadat následovně *GET /studenti/123*.

Reprezentace odpovědi na tento požadavek by pak v prvním přípdě, tedy formátu XML, vypadala například jako ve výpisu 3.1.

```
<student>
        <id>123</id>
        <jmeno>Jan</jmeno>
        <prijmeni>Novak</prijmeni>
        <login>xnovakYY</login>
</student>
```

Výpis 3.1: Odpověď reprezentovaná ve formátu XML

V případě druhém, kdy by odpověď byla reprezentována ve formátu JSON, by potom vypadala stejně jako ve výpisu 3.2.

```
{
        "student": {
                "id": "123",
                "jmeno":"Jan",
                "prijmeni":"Novak",
                "login":"xnovakYY"
        }
}
```

Výpis 3.2: Odpověď reprezentovaná ve formátu JSON

Kapitola 4

Sémantické vyhledávání

Jak zmiňují doktoři Vazquez a López-de-Ipiña ve svém článku o protokolu mRDP [10], sémantické vyhledávání je pro nás velmi důležité z hlediska výpočetních systémů a mobilních ad-hoc sítí, kde existující entity musí být neustále navzájem informovány a adaptovat stále se měnící síťovou topologii. Typickým dotazem pro sémantické vyhledávání může být například dotaz *Počítač v pracovně*, *Dokument napsán Petrem* nebo *Tiskárna poblíž počítače*. Nalezení takovýchto zdrojů nemusí být v dnešní době nijak obtížné. Stačí mít pouze správně uložené infromace o zdrojích (například v XML souboru) a při sestavování požadavku pouze správně vytvořit odpovídající dotaz.

Větší problém nám může nastat v případě, že chceme položit složitější dotaz nebo dokonce chceme být schopni vyřešit jakýkoliv neočekávaný dotaz. Uvažujme například následující rozložení zdrojů:

- **Kniha A** se nachází v **polici 13**.
- **Police 13** je umístěna v sekci **Odborné texty**.
- **Knihovna** má sekci **Odborné texty**.

Položme si nyní například dotaz typu *Nachází se v Knihovně Kniha A?*. Pro člověka je řešení tohoto problému triviální, avšak pro počítač už ne tak úplně. Musí mít někde v paměti definovanou celou strukturu zanoření zdrojů, a to může být především u rozsáhlejších struktur problém.

Hlavním problémem, který je zde potřeba vyřešit, je způsob jakým jsou dotazy vyhodnocovány a *přeloženy* na odpovídající kandidáty. K tomu, aby vůbec nějaké vyhodnocování bylo možné realizovat, se do značné míry využívá poznatků z oblasti tzv. *sémantického webu*, podpůrných jazyků, protokolů a nástrojů.

4.1 Resource Description Framework

Resource Description Framework (RDF) [8] je standartní model pro výměnu dat ve webovém prostředí. Jedná se o možnost rozšíření dat, běžně dostupných na webu, mimo standartního lidsky čitelného popisu o popis čitelný strojově. Tím, že mají data nyní i strojově čitelná metedata, nám vzniká možnost efektivnější výměny dat, vyhledávání, kategorizace, navigace, a tak dále.

Uvažujme nyní další sadu informací o zdrojích, jako doplněk k sadě použité v kapitole 4.

- **Kniha A** má kapitolu **RDF**.
- **Kniha B** se nachází v sekci **Politické knihy**.
- sekce **Odborné texty** se nachází vedle sekce **Politické knihy**.

To co nám RDF přináší, je nejen mechanismus pro uchovávání těchto informací o zdrojích, ale také možnost spojení několika sad informací do jedné. Spojením sady z kapitoly 4 a naší nově nadefinované sady, jsme najednou schopni říci, že kniha B se také nachází v Knihovně. Máme také více informací o Knize A:

- **Kniha A** má kapitolu **RDF**.
- **Kniha A** se nachází v **polici 13**.
- **Kniha A** se nachází v sekci **Odborné texty**.
- **Kniha A** se nachází v **Knihovně**.

Aby se informace o zdrojích daly uchovávat, dala se mezi nimi vytvořit vazba a dalo se nad nimi dobře vyhledávat, musí být tyto informace uloženy ve správném formátu. RDF využívá pro ukládání dat takzvané *RDF Tripples*. RDF Tripple si můžeme představit jako trojici podmětu, vlastnosti a předmětu. Pro informaci **Kniha A** má kapitolu **RDF** je potom RDF Tripple následující:

- **Podmět** – Kniha A
- **Vlastnost** – má kapitolu
- **Předmět** – RDF

4.2 Web Ontology Language

Web Ontology Language, dále jen OWL, je sémantický značkovací jazyk, který je určený pro práci s Ontologií. Pro lepší vysvětlení pojmu OWL, je třeba si představit pojem Ontologie.

Ontologie je specifická organizace znalostí v dané oblasti. Jejím cílem je organizovat a řadit znalosti. Pro vytvoření ontologie je nutné definovat rozsah znalostí, které je třeba zachytit a zpracovat [1].

Jak již bylo zmíněno, jazyk OWL slouží pro práci s Ontologií. Pokud tedy máme v Ontologii nějaké znalosti, tento jazyk nám poskytuje prostředky, abychom s nimi mohli pracovat. Umožňuje nám definovat, prezentovat a sdílet Ontologii. Jazyk OWL se používá pro aplikace, které potřebují informace nejen prezentovat lidem, ale i strojově zpracovávat. Byl vyvinut jako slovník, rozšiřující RDF [5]. V podstatě se tedy jedná o reprezentazaci znalostí ve formátu *RDF Tripple* pomocí značkovacího jazyka.

4.3 Sémantické vyhledávání v praxi

Jak funguje takové sémantické vyhledávání v praxi, je velmi dobře patrné na příkladu, který ve své práci představili doktoři Vazquez a López-de-Ipiña [10]. Konkrétně uvažujme firmu vyrábějící inteligentní vrtačku, která se automaticky vypne, pokud máme v místnosti

přítomnou hořlavou látku a vrtáme do železného materiálu. Vrtačka periodicky hledá přítomnost hořlavého materiálu a v případě, že jej nalezne, vypne se a vypíše na vestavěný display varovnou zprávu. Znalosti, které pro tuto svou funkcionalitu používá, jsou uloženy v kolektivním úložisti znalostí a jsou uvedeny ve výpisu 4.1.

```
<urn:uuid:vrtacka> loc:umistenV <urn:uuid:mistnost1>
<urn:uuid:vrtacka> task:vrtani <urn:uuid:povrch1>
<urn:uuid:povrch1> rdf:typ fur:ZeleznaPolice
<urn:uuid:box1> loc:obsahuje <urn:uuid:olej>
<urn:uuid:box1> loc:umistenV <urn:uuid:mistnost1>
<urn:uuid:olej> rdf:typ fuel:ChemickyOlej
```

Výpis 4.1: Vstupní znalosti

Existující ontologie v úložišti nám doplní vazby umístění předmětů a informace, které jsou uvedené ve výpisu 4.2.

```
fur:ZeleznaPolice rdf:podtyp fur:ZeleznyMaterial
fuel:ChemickyOlej rdf:podtyp fuel:HorlavyMaterial
<urn:uuid:olej> loc:umistenV <urn:uuid:box1>
<urn:uuid:olej> loc:umistenV <urn:uuid:mistnost1>
```

Výpis 4.2: Znalosti doplněné s využitím ontologie

Dotaz, který vrtačka periodicky pokládá pro zajištění své funkcionality, je potom znázorněn ve výpisu 4.3.

```
IDENTIFY ?material WHERE
       ?material loc:umistenV ?mistnost
       ?material rdf:typ mat:HorlavyMaterial
       <urn:uuid:vrtacka> loc:umistenV ?mistnost
       <urn:uuid:vrtacka> task:vrtani ?sur
       ?sur rdf:typ mat:ZeleznyMaterial
```

Výpis 4.3: Dotaz pro nalezení hořlavého materiálu

Pokud bychom předchozí dotaz položili bez jakékoliv podpory ze strany ontologie, tedy za využití pouze prostého aplikování striktního vyhodnocení na základě znalostí z výpisu 4.1, neodbrželi bychom žádný výsledek. Neměli bychom k dispozici žádné infromace o materiálech a doplnění vazeb umístění předmětů. Ovšem s využitím ontologie a kolektivního úložiště informací nám položený dotaz vrátí hledaný objekt (<urn:uuid:olej>).

Kapitola 5

Protokol mRDP

Návrh protokolu *Multicast Resource Discovery Protocol* (mRDP) byl poprvé představen v roce 2007. Jeho autoři doktoři Vazquez a López-de-Ipiña jej navrhli jako teoretický model, který nikdy nenaimplementovali. Tato kapitola, včetně podkapitol, do značné míry čerpá z jejich článku, vydaného v knize Computer Networks [10].

Tento protokol definuje následující důležité části a cíle:

- Aby bylo dosaženo flexibility, nesmí být použit žádný centrální server. Komunikace pro dotazy na zdroje bude typu *multicast*.
- Protokol bude podporovat dva základní režimy. Prvním bude plně sémantický mód s podporou sématického vyhledávání. Druhým pak mód základní, který bude podporovat pouze standardní operace známé z architektury REST. Základní mód je zde z důvodu přítomnosti výkonnostně slabších zařízení, které nemusí zvládat sémantické vyhledávání.
- Musí být k dispozici jazyk, který dokáže spolehlivě a efektivně vyhodnocovat sémantické dotazy.
- Ke správné funkcionalitě nestačí pouze být schopen určit o který zdroj se jedná, ale také najít jeho přesnou lokaci.
- Celá implementace protokolu musí být postavena nad TCP/IP infrastrukturou s integrací s protokolem HTTP z důvodu využití stávající webové architektury a HTTP zabezpečovacích mechanismů.

Výsledná funkcionalita tohoto protokolu by poté měla poskytovat pouze dvě rozdílné funkce:

- **Identifikace zdroje** – nalezení zdroje poskytovaného serverem, na základě dotazu, který splňuje určité podmínky. Například dotaz *Najdi všechny předměty umístěné v* ***urn:uuid:mistnost1***.
- **Umístění zdroje** – nalezení umístění zdrojů, které obsahují informace o námi hledaném zdroji. Například dotaz *Kde najdu informace o zdroji* ***urn:uuid:olej****?*.

Popis první zmíněné funkce byl již přiblížen v kapitole [**?**]. Pokud se blíže podíváme na návrh druhé zmíněné funckionality, je koncept o něco jednodušší. Klient odešle požadavek

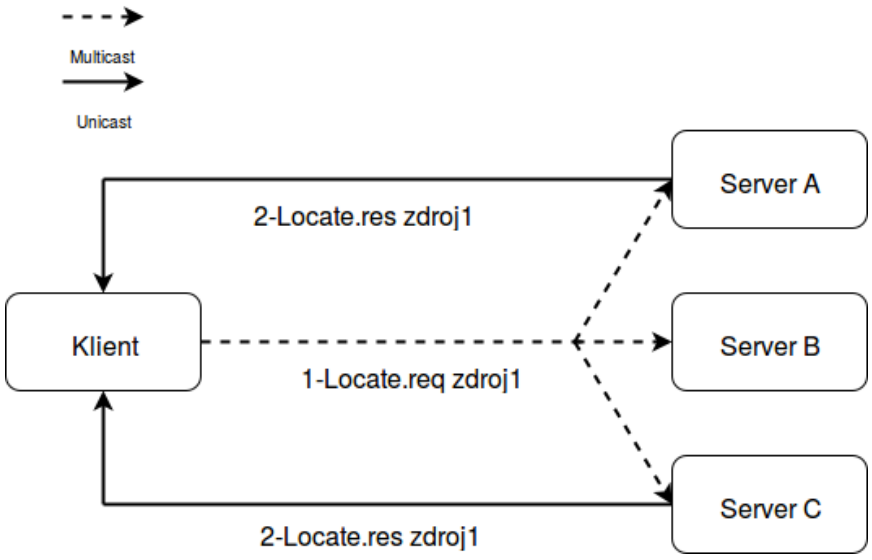
s dotazem na nalezení informací o konkrétním zdroji (například dotaz typu *Dej mi všechny informace o http://studenti.cz/xnovak99*). Každý mRDP server obsahující byť jen malou část informací o hledaném zdroji, odpoví se zprávou obsahující URL adresu, kde mohou být informace o zdroji nalezeny. V odpovědi na dotaz se nachází pouze URL adresa, namísto přímého posílání informací o hledaném zdroji. Tento formát odpovědi byl zvolen hned z několika důvodů:

- Odlehčení provozu na síti. Pakety obsahující pouze URL, jsou mnohem menší než ty, které by nesly data o zdroji.
- Rozhodnutí o tom zda si data o hledaném zdroji klient stáhne či nikoliv, je necháno pouze na jeho rozhodnutí.
- URL adresu můžeme použít vícekrát za předpokladu, že se adrese zdroje nezmění.

5.1 Komunikující entity v mRDP

Protokol mRDP představuje dvě entity: mRDP klient a mRDP server. Klient mRDP je klientská stanice, která posílá dotazy a snaží se identifikovat hledaný zdroj, či najít umístění zdroje. Klientské stanice komunikují v rámci protokolu mRDP výhradně multicastově za pomoci nespolehlivého protokolu mRDP. Server mRDP je potom entita, která přijímá dotazy od mRDP klienta a snaží se je vyhodnotit, případně odpovědět, zda se informace o hledaném zdroji u něj nachází. Komunikace u serverových entit, narozdíl od klientských, probíhá ve formě unicastových zpráv. Protože odpovědi mohou být značně větší než dotazy (uvažme například dotaz typu *Dej mi všechny informace o všech studentech*), využívá se protoklu TCP, který nám zajistí fragmentaci a spolehlivost.

Na následujícím obrázku je ukázka, jak probíhá komunikace mezi mRDP klientem a mRDP servery v případě hledání umístění zdroje. Informace o hledaném zdroji mají dva mRDP servery a tak odpovídají oba dva.



Obrázek 5.1: Ukázka nalezení umístění zdroje v mRDP.

5.2 Zabezpečená komunikace v rámci protokolu mRDP

Dotazy, které klient odesílá na mRDP servery, musí být čitelné jakýmkoliv serverem. Tento požadavek je důležitý především pro podstatu samotného výhledávání, protože dopředu netušíme, které servery budou naši multicastovou zprávu s dotazem číst. Z toho důvodu nemusí být dotaz vyslaný od klienta zašifrovaný. U odpovědí je ale situace jiná. Server přesně ví, s kterým klientem komunikuje, a tak je možné a dokonce žádoucí, aby byla komunikace zabezpečena proti případným útokům. Návrh řešení tohoto problému představili doktoři Vazquez a López-de-Ipiña ve svém článku o protokolu [10]. Pro dosažení požadovaného zabezpečení, je využito protkolu HTTPS. Klient do zprávy s dotazem vloží mimo samotného dotazu i *HTTP URI Callback* a tak každý mRDP server může odpověď zaslat na tuto adresu s využitím zabezpečovacích mechanismů, které nám HTTPS nabízí. Výhody využití HTTPS protokolu jsou zřejmé:

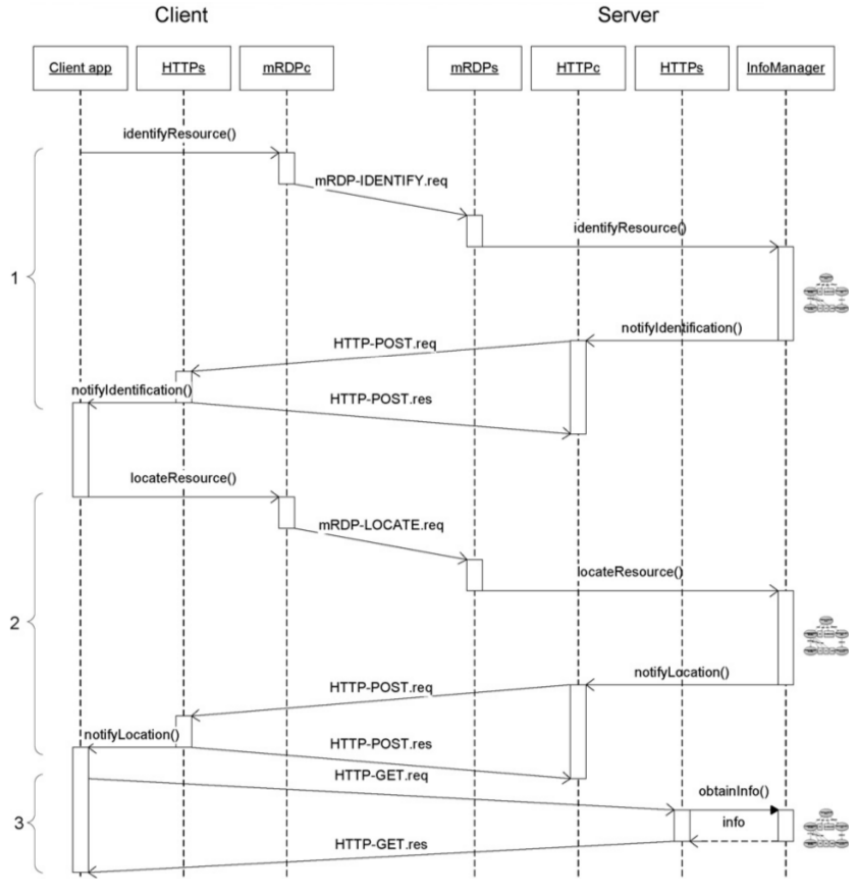
- Protokol HTTPS je ve stávající architektuře Internetu hojně využíván a není tedy potřeba žádných speciálních zásahů do této architektury.
- HTTPS již obsahuje zabezpečovací mechanismy (ať se jedná o HTTP basic nebo digest, případně samotného HTTPS), které mohou být lehce použity.
- HTTPS využívá protokolu TCP a je tedy zaručena spolehlivost a neomezená velikost zasílaných zpráv.

Jak tedy celá komunikace probíhá je nejlépe vidět na následujícím příkladu s UML diagramem 5.2, který je převzat z článku o představení protokolu mRDP [10]. Jednotlivé kroky prováděné v tomto diagramu jsou následující:

1. Klientská aplikace (resp. mRDP klient) odešle multicastový požadavek do sítě na identifikace zdroje. Server, který tuto zprávu odbrží a následně identifikuje zdroj, vygeneruje a pošle zpět klientovi HTTP Post požadavek.
2. Pokud klient potřebuje o již identifikovaném zdroji více informací, odešle do sítě opět multicastovou zprávu s požadavkem na nalezení umístění zdroje. Servery vyhodnotí zda mají informace o tomto zdroji či nikoliv. V případě, že server nějaké informace má, vygeneruje HTTP Post požadavek obsahující URL daných informací.
3. Klient si může stáhnout informace o hledaném zdroji prostřednictvím jednoduchého HTTP Get požadavku na URL, kterou obdržel od serveru.

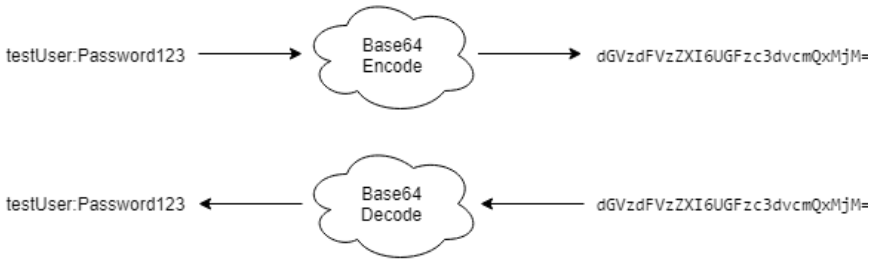
5.2.1 Basic access authentication

Aby se komunikace dala nazvat zabezpečenou, potřebuje knihovna ještě možnost ověření uživatele. K tomuto účelu je postačující technologie `Basic access authentication`, nicméně pouze za předpokladu, že je využit protokol HTTPS. Tento mechanismus totiž odesílá kromě běžných HTTP hlaviček ještě navíc hlavičku `Authorization`, ve které jsou uložena data sloužící pro ověření uživatele. Tyto data se samozřejmě před odesláním do sítě ještě zašifrují pomocí technologie `Base64` což nám ale neposkytne dostatečnou úroveň zabezpečení protože Base64 hash je zpětně dekódovatelný 5.3. Proto je zde vstupní podmínka, že musí být využit protokol HTTPS, kde je bezpečná už komunikace sama o sobě.



Obrázek 5.2: Příklad komunikace podle protokolu mRDP (Převzato z [10]).

Pokud tedy je například uživatel který mý přihlašovací jméno `testUser` a heslo `Password123`, musí se nejdříve sestavit Base64 hash těchto informací. Ten sestavíme tak, že spojím přihlašovací jméno a heslo pomocí oddělovače `:` a provedeme zakódování pomocí technologie Base64. Graficky znázorněná ukázka této operace včetně ukázky dekódování je na obrázku 5.3.



Obrázek 5.3: Příklad kódování a dekódování pomocí technologie Base64.

Tento hash je potom odeslán v hlavičce HTTPS požadavku a na straně serveru dojde k dekódování přihlašovacích údajů. Pomocí těchto údajů server může ověřit, zda se skutečně jedná o daného uživatele.

5.3 Vyhodnocování sémantických dotazů

Servery, které budou vyhodnocovat sémantické dotazy od mRDP klientů, podporují dva jazyky pro vyhodnocování. Jsou to jazyky SPARQL a Plant [11]. Ukázka toho jakým způsobem vyhodnocuje jazyk SPARQL, byla předvedena v kapitole o sémantickém vyhledávání 4.3. Z důvodu vysoké náročnosti na výkon serveru je nutné využívat i dalšího jazyku Plant. Tento jazyk je rozšířením standradního jazyka **N-Triples** a má velmi jednoduchou syntaxi. Skládá se ze třech následujících částí:

```
predmet = URL | IdZdroje | promenna
objekt = URL | IdZdroje | literal | promenna
promenna = '?' jmeno
```

Výpis 5.1: Prvky jazyka Plant

Tento jazyk nám tedy dovolí zdroje popisovat. Nicméně oproti jazyku SPARQL neposkytuje logiku pro vyhodnocování dotazů. Tuto logiku tedy musíme naiplementovat pro účel naší knihovny sami za pomocí algoritmu **The Plant Query Resolution Algorhitm**, který byl uveden v článku o protokolu mRDP [10]. Jednotlivé kroky tohoto algoritmu jsou následující:

1. Pokud již tak není, namapuj dvojice atribut-hodnota do tvaru odpovídajícímu RDP Triples.
2. Identifikuj všechny proměnné v dotaze.
3. Pro každý Plant vzorec vyber z informačního modelu trojice, které tento vzorec splňují a označ kombinaci platných hodnot proměnných poskytnutých trojicí (* pro libovolnou hodnotu).
4. Nahraď * za pomoci dostupných hodnot příslušné proměnné v kombinaci s ostatními Plant vzorci.
5. Identifikuj kombinace hodnot, které odpovídají všem vzorcům. Tyto kombinace představují řešení pro proměnné. Pokud žádná kombinace nebyla identifikována, dotaz nemá řešení.

5.4 Formát mRDP zpráv

Vzhledem k tomu, že naimplementovaná knihovna má být k dispozici pro co nejvíce zařízení, budeme pro následující popis formátu zpráv předpokládat nejméně omezující způsob řešení, a to zprávy nesoucí dotaz v jazyce Plant. Formát zprávy je popsán v podobě, v jaké byl navržen při představení protokolu mRDP [10].

Požadavky protokolu mRDP jsou odeslány za pomocí protokolu UDP na multicastovou adresu 224.0.24.1 a UDP port 2773. MIME typ zprávy pro Plant dotazy je *application/-com.awareit.plant*. Zpráva by měla mít prostor pro přídavné hlavičky, které budou moci v budoucnu rozšiřovat funkcionalitu protokolu mRDP. Požadavek mRDP se skládá z tzv. *Request line*, nula nebo více hlaviček a volitelného těla zprávy. Request line je hlavním prvkem požadavku mRDP a obsahuje typ funkce, kterou po mRDP serveru požaduje. Příkaz **IDENTIFY** je následován vždy proměnnou, kterou má dotaz vyřešit a příkaz **LOCATE** URL adresou zdroje.

Hlavičky jsou ve formátu stejném jako tradiční hlavičky protokolu HTTP. Pro mRDP byly definovány následující čtyři hlavičky:

- **NSeq** – pořadové číslo požadavku pro detekci duplicity zprávy od klienta a spárování požadavků a odpovědí
- **Content-type** – MIME typ těla zprávy. K dispozici je na výběr ze dvou typů: *application/com.awareit.plant* a *application/sparql-query*. Tato hlavička je nepovinná pokud zpráva nemá tělo (dotaz na nalezení umístění zdroje).
- **Content-length** – délka těla zprávy v bajtech. Tato hlavička je nepovinná, pokud zpráva nemá tělo.
- **Callback-URI** – URI adresa, na kterou má mRDP odeslat odpověď

V ukázce 5.2 je uveden příklad jednoduché mRDP zprávy, která odesílá požadavek na nalezení umístění informací o zdroji *olej*. Vzhledem k tomu, že jsou tyto zprávy zasílány pomocí protokolu UDP, který je nespolehlivý, realizujeme odeslání samotné zprávy celkem třikrát.

```
1. LOCATE urn:uuid:olej mRDP/1.0
2. NSeq 3
3. Callback-URI:http://192.168.1.113/mrdpcallback
```

Výpis 5.2: mRDP zpráva pro nalezení informací o zdroji

Server mRDP po vyřešení dotazu sestaví odpověď pro mRDP klienta, která je reprezentována HTTP požadavkem na URI adresu, která byla specifikována v požadavku v hlavičce *Callback-URI*. Pro tuto odpověd navrhli doktoři Vazquez a López-de-Ipiña jazyk *ReDEL*, neboli Resource Description Endpoints Language. V ukázce této zprávy 5.3, do značné míry převzaté z [10], můžeme vidět, že řádek 13 - 15 obsahuje informace o hledaném zdroji. Konkrétně pak mohou být informace, které hledáme staženy pomocí HTTP Get požadavku z adresy uvedné v location url.

```
1. POST /mrdpcallback HTTP/1.0
2. Host: 192.168.1.113
3. NSeq: 3
4. Content-type: application/com.awareit.redel+xml
5. Content-length: 341
6.
7. <?xml version="1.0" encoding = "UTF-8"?>
8. <redel xmlns="http://www.awareit.com/soam/2006/04/redel"
9.   xmlns:xsi="http://w3c.org/2001/schema/XMLSchema-instance"
10.  xsi:schemaLocation="http://www.awareit.com/soam/2006/04/redel
11.  http://www.awareit.com/soam/2006/04/redel.xsd">
12.
13. <resource uri="urn:uuid:olej">
14. <location url="http://192.168.1.199/olej_desc.rdf"
       type="http://www.awareit.com/soam/2006/04/srdfws#httpGet"/>
15. </resource>
16.
17. </redel>
```

Výpis 5.3: Odpověď od mRDP serveru v jazyce ReDEL

Kapitola 6

Implementace knihovny

6.1 Balíky knihovny

Vlastní implementace knihovny je rozdělena do několika logických balíků podle odpovídajích odpovědností. Odpovědnosti balíku byly logicky navrženy za účelem co nejrestriktivnější viditelnosti tříd v nich umístěných. Jednotlivé balíky jsou představeny v následujícím výčtu.

- **api** – balík, poskytující veřejné API knihovny, prostřednictvým kterých se k jejím funkcím přistupuje
- **cache** – balík, který v sobě obsahuje třídy pro krátkodobé ukládání údajů o zprávách a klientech
- **communication** – balík, zajišťující síťovou komunikaci mezi jednotlivými zařízeními používající knihovnu OpenmRDP.
- **exceptions** – balík vyjímek
- **logger** – balík, ve kterém jsou umístěny třídy pro zaznamenávání událostí
- **messageprocessors** – balík pro zpracování příchozích zpráv
- **messages** – balík, obsahující základní třídy pro sestavení zpráv
- **model** – balík, ve kterém jsou uloženy třídy zajišťující informační model. Je zde implementována funkcionalita informační báze, ontologie a informačního manažera.
- **query** – balík, zajišťující funkcionalitu spojenou s vyhodnocováním dotazů
- **security** – balík, nesoucí v sobě třídy použité pro zabezpečení
- **server** – balík čistě serverových tříd. Nachází se zde například třídy pro obsluhu HTTP požadavků a konfigurační třídy.

6.2 Vyjímky

Pro svou implementaci knihovna openmRDP využívá několik typů nově naimplementovaných vyjímek. Některé z vyjímek jsou propagovány i mimo knihovnu, prostřednictvím veřejných metod rozhraní a programátor, implementující knihovnu OpenmRDP, musí být

připraven je korektně ošetřit. Konkrétní vyjímky i s účelem jejich vytvoření jsou popsány v následujícím výčtu.

- **AddressSyntaxException** – Hlídaná vyjímka, která se využívá při zpracování adres zpráv. Pokud tato vyjímka nastane, je chyba zásádní, protože knihovna neví na jakou adresu má zaslat odpoveď.
- **InformationBaseException** – Běhová vyjímka, která může nastat při načítání informační báze z XML souboru. Tato vyjímka není kritická a v případě, že nastane, knihovna pokračuje dále ve své činnosti s prázdnou bází znalostí.
- **MessageDeserializeException** – Běhová vyjímka, která může nastat při převádění přijatého řetězce na objekt typu BaseMessage 6.3. Pokud tato vyjímka nastane, je přijatá zpráva ignorována a knihovna čeká na další zprávu.
- **NetworkCommunicationException** – Hlídaná vyjímka, která značí chybu v síťové komunikaci. Je propagována prostřednictvím veřejného API mimo implementaci knihovny a záleží na programátorovi, který knihovnu implementuje, jak ji ošetří.
- **OntologyException** – Běhová vyjímka, značící chybu při načítání dat z XML souboru s ontologií. Podobně jako vyjímka InformationBaseException, není tato vyjímka kritická a knihovna může pro svůj běh využít výchozí hodnoty.
- **QueryProcessingException** – Běhová vyjímka používaná při vyhodnocování dotazů. V případě jejího nastání, je tato událost zaznamenána prostřednictvím logů.
- **QuerySyntaxException** – Hlídaná vyjímka, která nastává při špatné syntaxi dotazu. V případě, že je vyvolána, vrací knihovna stejnou odpoveď, jako když se nepodaří zdroj identifikovat.

6.3 Zprávy

Knihovna openmRDP využívá několik typů zpráv, které si postupně všechny představíme. Všechny zprávy, vyjma zprávy `Connection Information`, vychází ze základní implementace třídy `BaseMessage`, skládající se z operačního řádku, mapy hlaviček a těla zprávy. Pro snadnou tvorbu všech typů zpráv je v balíku `messages` k dispozici třída `MessageFactory` implementující návrhový vzor `Factory`. Využití tohoto návrhového vzoru nám zajistí, že vytvořené zprávy jsou vždy v konzistentním stavu, připraveny k odeslání. Všechny objekty, reprezentující zprávy, jsou zároveň neměnitelné, takže je minimalizována chybovost při manipulaci s nimi.

6.3.1 Operační řádek

První informací, kterou zpráva nese, je operační řádek. Skládá se z typu operace (určuje zda se jedná o zprávu typu texttttLOCATE, `IDENTIFY` nebo `POST`), jména zdroje (ať již se jedná o zdroj, který chceme lokalizovat, nebo o proměnnou, kterou chceme identifikovat) a protokolu, který je využit (`HTTP`, `HTTPS` nebo `mRDP`). Implementace operačního řádku se nachází ve třídě `OperationLine` a kromě obsahu těchto základních údajů nemá žádnou další funkcionalitu.

6.3.2 Tělo zprávy

Tělo zprávy je nepovinným atributem třídy `BaseMessage`, který je reprezentován objektem typu `MessageBody`, a skládá se ze samotného obsahu a atributu, který údává typ obsahu. Tento obsah může nabývat následujících tří základních typů.

- **PLANT query** – pro dotaz v jazyce PLANT
- **SPARQL query** – pro dotaz v jazyce SPARQL
- **REDEL** – pro odpověď v jazyce REDEL

Tato třída nám také poskytuje možnost vypočítat délku zprávy, kterou používáme pro jednu z hlaviček zpráv, a to konkrétně `Content-Length`. Tento výpočet probíhá pouze převodem těla zprávy na bajtové pole a následným určením jeho délky, přičemž za standartní kódování je zde defaultně považováno `UTF-8`.

6.3.3 Zpráva typu LOCATE

Asi nejjednodušší zprávou na tvorbu je zpráva typu `LOCATE`. Tato zpráva neobsahuje žádné tělo a nese v sobě pouze dvě hlavičky zprávy. Operační řádek se tedy skládá z typu operace LOCATE, jména zdroje, který chceme lokalizovat a využitého protokolu mRDP. Co se hlaviček zprávy týká, obsahuje pouze hlavičku sekvenčního čísla a takzvaného Callback-URI, které je popsáno v kapitole 5.4.

6.3.4 Zpráva typu IDENTIFY

Druhým typem zprávy je zpráva typu `IDENTIFY`. Obsah je oproti předchozí zprávě rozšířen o několik dalších hlaviček a tělo zprávy `MessageBody`. Operační řádek obsahuje typ operace IDENTIFY, jméno zdroje, které je v tomto případě nahrazeno proměnnou, kterou chceme identifikovat a využitý protokol, který zůstává stejný jako v případě zprávy `LOCATE`. Vzhledem k tomu, že tělo zprávy již není prázdné, musíme přidat dvě další hlavičky, a to konréktně hlavičku `Content-Type` a `Content-Length`.

Hlavička `Content-Type` v našem případě obsahuje hodnotu `application/com.aware-it.plant` a obsah hlavičky `Content-Length` je dopočítán z těla zprávy.

6.3.5 Zpráva typu ReDEL

Dalším typem zprávy je zpráva typu ReDEL. Slouží výhradně k odpovědím, které server odesílá zpět klientovi. Zpráva využívá protokolu HTTP, a proto se její operační řádek skládá z typu operace POST, jména zdroje, které je zde reprezentováno takzvaným endpointem zařízení, na které je odpověď odeslána, a potokolu HTTP. Hlavička `Content-Type`, `Content-Length` a hlavička nesoucí sekvenční číslo zůstává stejná jako u předchozího typu zprávy. Nově je mezi hlavičkami zprávy typu ReDEL hlavička typu `HOST`, která obsahuje adresu klienta, který má obdržet odpověď. Tělo zprávy nese samotnou odpověď, zapsanou v jazyce ReDEL.

6.3.6 Zpráva typu Connection Information

Posledním typem zpráv, které knihovna definuje, je zpráva Connection Information a je využívána v případě, že server chce oznámit klientovi, že pro něj má k dispozici dostupné

informace. Tato zpráva slouží k předání informací, za pomoci kterých se klient může k serveru připojit a komunikovat s ním. Informace uložené ve zprávě Connection Information jsou uvedeny v následujícím výčtu.

- **SERVER** – adresa serveru, ke kterému se klient má připojit
- **NSEQ** – sekvenční číslo zprávy
- **PROTOCOL** – protokol, pomocí kterého bude server s klientem komunikovat (HTTP nebo HTTPS)
- **AUTHORIZATION** – informace o tom, zda server vyžaduje autorizaci klienta či nikoliv (příznak REQUIRED nebo NONE)

Ukázka toho, jak vypadá zpráva Connection Information pro server, který očekává nezabezpečenou HTTP komunikaci na adrese `http://192.168.1.53:24471/auth`, je uvedena v úkazce 6.1.

```
1. SERVER: 192.168.1.53:24471/auth
2. NSEQ: 2
3. PROTOCOL: HTTP
4. AUTHORIZATION: None
```

Výpis 6.1: Příklad zprávy typu Connection Information

6.3.7 Serializace a deserializace zpráv

Vzhledem k tomu, že zprávy, které obdržíme prostřednictvím sítě, jsou v podobě bajtového pole, musíme je převést na odpovídající objekt typu BaseMessage. K tomuto účelu nám slouží třídy `MessageSerializer` a `MessageDeserializer`. Obě třídy nemají žádnou jinou funkcionalitu než převod zpráv mezi bajtovým polem a obejktem typu BaseMessage, popřípadě objektem typu Connection Information, a naopak.

6.4 Funkce LOCATE

Jednou ze dvou hlavních funkcí knihovny openmRDP je funkce LOCATE. V následujících podkapitolách jsou popsány prerekvizity nutné pro proces nalezení lokace zdroje i samotný algoritmus nalezení lokace. Před samotným procesem hledání lokace je třeba načíst a sestavit bázy znalostí a ontologii, kterou bude knihovna využívat. Stejně tak je potřeba instanciovat manažera informací, který v celém procesu hledání lokace hraje klíčovou roli. Po úspěšné inicializaci všech prerekvizit je hlavním klíčem ke správnému nalezení lokace zdroje takzvaný strom lokací 6.4.5, po jehož sestavení již není samotná lokalizace zdroje nijak výpočetně náročná.

6.4.1 Báze znalostí

Knihovna OpenmRDP využívá pro oba své módy (**IDENTIFY**, **LOCATE**) bázi znalostí. Informace, které báze poskytuje, jsou uloženy v XML souboru pojmenovaném jako `informationBase.xml` a přístup k nim je ve vlastní implementaci umožněn prostřednictvím služby `InformationBaseService`. Tato služba poskytuje několik základních operací, a to

konkrétně načtení všech informací ze souboru, přidání nové informace do souboru a odstranění informace ze souboru. Takto pojmenovaný soubor je knihovnou načítán z uživatelského domovského adresáře, nastaveného v systémové proměnné pod názvem `user.dir`. Příklad XML souboru, ve kterém je báze informací uložena, je uveden v ukázce 6.2

```
<?xml version="1.0" encoding="UTF-8"?>
<InformationBase>
	<Information>
		<Subject>urn:uuid:room1</Subject>
		<Predicate>&lt;loc:locatedIn&gt;</Predicate>
		<Object>urn:uuid:building1/Object>
	</Information>
	<Information>
		<Subject>urn:uuid:room1</Subject>
		<Predicate>&lt;loc:contains&gt;</Predicate>
		<Object>urn:uuid:box1</Object>
	</Information>
	<Information>
		<Subject>urn:uuid:fuel1</Subject>
		<Predicate>&lt;loc:locatedIn&gt;</Predicate>
		<Object>urn:uuid:box1</Object>
	</Information>
</InformationBase>
```

Výpis 6.2: Báze znalostí uložená v XML souboru

6.4.2 Ontologie

K tomu, aby byly informace z informační báze kompletní a dalo se z nich čerpat při vyhodnocování dotazů, je vhodné je doplnit o znalosti z ontoligie. Ontologie nám v tomto případě udává sémantiku predikátů informací. Minimální údaje uvedené v ontolgii, by měly udávat který predikát je použit pro zanoření o úroveň níže a který pro vynoření o úroveň výše ve stromu lokací 6.4.5. Pokud tyto informace v ontologii uvedeny nejsou, použije se výchozí hodnota `<loc:contains>` pro zanoření a `<loc:locatedIn>` pro vynoření.

Další důležitou informací, která je v ontologii také uložena, je oddělovač. Pomocí oddělovače se bude sestavovat výsledná cesta umístění zdroje. Pokud v ontologii tato informace uvedena není, bude použit standartní oddělovač \.

Dále může ontologie nepovinně obsahovat údaje o tranzitivních predikátech. Tranzitivní predikáty nám doplní informační model o informace, které z těchto tranzitivních predikátů vyplývají. Pro lepší pochopení této problematiky si uvedeme příklad. Nechť naše informační báze obsahuje následující informace.

- **urn:uuid:surface1 rdf:type fur:steelShelf**
- **fur:steelShelf rdf:subtype mat:metallicThing**

Z ontologie pak máme následující údaj o tranzitivním predikátu.

- **rdf:type rdf:subtype**

Spojíme-li tyto zmíněné poznatky dohromady, jsme schopni do informační báze doplnit jednu další novou informaci a to **urn:uuid:surface1 rdf:type mat:metallicThing**.

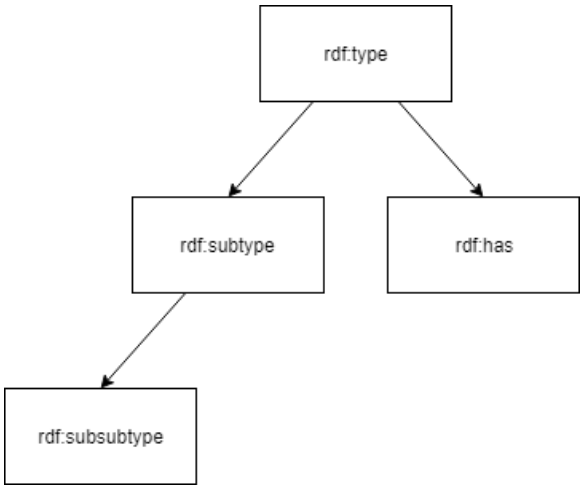
Ontologie je ve vlastní implementaci knihovny OpenmRDP uložena v podobě XML souboru pojmenovaném `ontology.xml` a přístup k údajům z ní je zajištěn za pomocí služby `OntologyService`. Umístění tohoto zdroje je stejné jako u souboru `informationBase.xml` a příklad takového souboru je uveden v ukázce 6.3

```
<?xml version="1.0" encoding="UTF-8"?>
<Ontology>
	<LevelUpPredicate>&lt;loc:locatedIn&gt;</LevelUpPredicate>
	<LevelDownPredicate>&lt;loc:contains&gt;</LevelDownPredicate>
	<Delimiter>/</Delimiter>
	<TransitivePredicates>
		<TransitivePredicate>
			<Parent>rdf:type</Parent>
			<TransitiveRelation>rdf:subtype</TransitiveRelation>
		</TransitivePredicate>
	</TransitivePredicates>
</Ontology>
```

Výpis 6.3: Ontologie uložená v XML souboru

6.4.3 Doplnění informační báze o nové informace

Samotné doplňování informační báze o nové informace má několik fází. Nejdříve se pro všechny informace, které slouží k určování lokace zdroje, vytvoří nové tak, aby splňovaly relaci symetrie. Dále musíme sestavit strom tranzitivních predikátů, který je reprezentován třídou `TransitivePredicateTree`. Protože tranzitivní predikáty je možné řetězit, může výsledný strom být podobný například stromu na obrázku 6.1.



Obrázek 6.1: Strom tranzitivních predikátů

S využitím tohoto stromu dále sestavíme seznam nejdelších možných tranzitivních predikátů. Pro strom uvedený na obrázku 6.1 by seznam vypadal následovně.

- rdf:type -> rdf:subtype -> rdf:subsubtype
- rdf:type -> rdf:has

Nyní, když je úspěšně načtena báze znalostí a načtena ontologie, máme všechny potřebné prerekvizity pro samotné vytváření nových pravidel, které probíhá pomocí následujícího algoritmu.

1. Pro každou položku ze seznamu tranzitivních predikátů vem první predikát `x`.
2. Z informační báze najdi všechny informace, které obsahují predikát `x`.
3. Pro každou nalezenou informaci z kroku 2 a pro každý další predikát ze seznamu tranzitivních predikátů, najdi v informační bázi takovou informaci, aby se dala sestavit nová, za použití subjektu z informace z kroku 2, predikátu ze seznamu tranzitivních predikátů a objektu z právě nalezené informace.

Nově vytvořené informace jsou spojeny s informacemi načtenými ze souboru `informationBase.xml` a jsou poskytnuty manažerovi informací jako kompletní báze informací.

6.4.4 Manažer informací

Klíčovou komponentou celé knihovny je manažer informací, který je naimplementován ve třídě `InfoManager`. Mezi základní funkcionalitu této třídy patří udržování informací o bázi znalostí, přidávání nových informací do báze znalostí či odstraňování zastaralých informací z báze znalostí, a to včetně propsání až do zdrojového XML souboru, ověřování pravdivosti informací a nalezení umístění konkrétního zdroje podle jeho názvu. K poslední zmíněné funkcionalitě, tedy hledání umístění zdroje, si manažer informací musí předem sestavit strom lokací. Pro jeho sestavení využívá již sestavené báze znalostí a z ontologie načtených predikátů zanoření a vynoření. Detailněji bude strom lokací popsán v následující kapitole 6.4.5. Manažer informací je jedna z prvních tříd, která je instanciována společně s veřejným API knihovny a z důvodu náročnější inicializace je pro tuto třídu využit návrhový vzor `Singleton` a tudíž se využívá po celou dobu běhu knihovny pouze jedna instance této třídy.

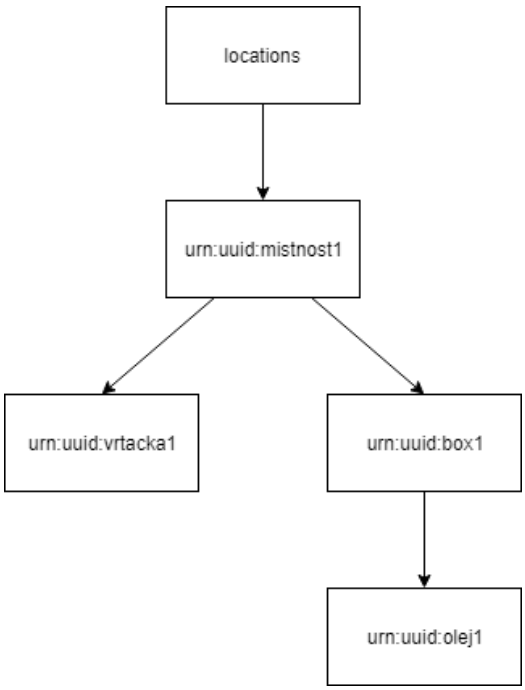
6.4.5 Strom lokací

Strom lokací si v implementaci knihovny OpenmRDP udržuje graf lokací všech dostupných zdrojů, který se sestavuje na základě informací z báze znalostí a predikátu pro zanoření ve stromu. Přístup k lokacím zprostředkovává služba `LocationTreeService`, která umožňuje strom vytvářet, a hledat umístění konkrétního zdroje. Mimo hledání umístění tato služba umožnuje také přidávat nebo odebírat informace o umístění do již existujícího stromu lokací. Služba `LocationTreeService` si udržuje instanci `LocationTree` po celou dobu své existence z důvodu náročnější inicializace této instance. Samotná inicializace probíhá za pomocí následujícího algoritmu.

1. Vytvoř strom s kořenem `locations`.
2. Najdi všechny lokace nejvyšší úrovně a přidej je do první úrovně pod kořen stromu.
3. Najdi všechny koncové uzly ve stromu.

4. Najdi v bázi znalostí všechny informace o umístění potomků pro uzly nalezené v kroku 3.
5. Přidej do stromu informace o umístění nalezené v kroku 4, pokud se již takové informace ve stromu nachází, přepiš je.
6. Opakuj algoritmus od kroku 3, dokud se přidávají nové informace do stromu nebo se strom mění.

Podle výše zmíněného algoritmu by strom lokací, sestavený pro příklad uvedený v kapitole 4.3, vypadal jako strom, který je znázorněn na obrázku 6.2.



Obrázek 6.2: Strom lokací

6.4.6 Nalezení lokace zdroje

Nalezení lokace zdroje, neboli funkce `LOCATE`, je jednou z klíčových funkcionalit knihovny OpenmRDP. Nyní, když máme sestaven strom lokací, můžeme vyhledat jakýkoliv zdroj, o kterém v něm máme uloženy informace o umístění. Stačí pouze rekurzivně prohledávat strom směrem od kořene k listům a v případě, že nalezneme uzel, obsahující jméno hledaného zdroje, sestavíme cestu k tomuto uzlu. Pro sestavení cesty je použit oddělovač, který byl načten z ontologie 6.4.2, případně výchozí.

Chceme-li například získat lokaci zdroje `urn:uuid:olej1` z příkladu, který je uveden v kapitole 4.3, s využitím stromu lokací z předchozí kapitoly 6.2, bude vypadat přesně následovně.

```
urn:uuid:mistnost1/urn:uuid:box1/urn:uuid:olej1
```

V případě, že o hledaném zdroji není ve stromu lokací uvedena žádná informace o jeho umístění, vrátí služba zpřístupňující tento strom hodnotu `null`.

Veškerá doposud zmíněná implementace zajišťuje první ze dvou základních funkcionalit knihovny OpenmRDP, kterou je funkce `LOCATE`.

6.5 Funkce IDENTIFY

Druhou, pro knihovnu OpenmRDP stěžejní, funkcionalitou je funkce IDENTIFY. V následujících podkapitolách jsou popsány prostředky, které se při vyhodnocování dotazů používají, a na závěr i samotný algoritmus, kterým k vyhodnocení dojde. Celý proces vyhodnocení může být dost náročný a je do velké míry ovlivněn složitostí dotazu a počtem proměnných v něm. Samotná velikost informační báze je samozřejmě také dalším činitelem, který ovlivňuje náročnost vyřešení dotazu. Pokud na základě dotazu a báze znalostí není algoritmus schopen zdroj identifikovat, vrátí prázdnou informaci o nalezeném zdroji. V současné implementaci knihovny OpenmRDP platí, že pokud nestihne být dotaz úspěšně vyhodnocen do deseti vteřin, považuje klient svůj dotaz za neúspěšný a již dál neočekává žádnou odpověď od serveru.

6.5.1 Podpůrné prostředky pro vyhodnocování

Pro proces vyhodnocování dotazů typu IDENTIFY, využívá knihovna OpenmRDP několik pomocných tříd. Jednou z hlavních tříd je již dříve zmíněná třída `InfoManager` s pomocí které ověřujeme, zda-li je informace pravdivá (nachází se v bázi informací) nebo hledáme potřebné informace. Další podpůrnou třídou je třída `QueryVariable`, která slouží pro uložení jména proměnné a případných možných hodnot, kterých může proměnná nabývat. Poslední podpůrnou třídou je potom samotná `Query`. Tato třída je využita k uložení množiny jednotlivých RDF trojic a příznaku, o jaký typ dotazu se jedná (PLANT, SPARQL).

6.5.2 Zpracování a vyhodnocení dotazu

Poté, co je příchozí zpráva deserializována do objektu typu `BaseMessage` a je z operačního řádku rozhodnuto, že se jedná o dotaz typu IDENTIFY, knihovna předá řízení třídě pro zpracování zpráv (`IdentifyMessageProcessor`). Tato třída využívá pro vyhodnocení dotazu třídu `QueryResolver` a po samotném vyhodnocení, pro finální vytvoření zprávy s odpovědí, již popsanou třídu `MessageFactory`.

QueryResolver jako první svou činnost provede rozložení zprávy na jednotlivé RDF trojice a vytvoří objekt typu Query. V dalším svém kroku identifikuje všechny proměnné nacházející se v dotazu a vytvoří množinu objektů typu QeuryVariable. Identifikace proměnných je v našem případě velmi triviální. Stačí pouze najít všechny unikátní subjekty a objekty, které začínají znakem '?'.

V dalším kroku najde QueryResolver pro každou RDF trojici ze vstupního dotazu odpovídající množinu potencionálně platných informací z báze informací. Pokud nastane situace, kdy některá trojice ze vstupního dotazu nebude mít žádnou potencionálně platnou informaci, QueryResolver vyhodnotí dotaz tak, že zdroj nenalezl. Informace se může stát potenciálně platnout v následujích případech.

- RDF trojice ze vstupu neobsahuje proměnnou a shoduje se s některou z informací z báze informací.
- RDF trojice ze vstupu obsahuje proměnnou ve svém subjektu a shoduje se s některými informacemi z báze informací v predikátu a objektu.

- RDF trojice ze vstupu obsahuje proměnnou ve svém objektu a shoduje se s některými informacemi z báze informací v subjektu a predikátu.
- RDF trojice ze vstupu obsahuje proměnné ve svém subjektu i objektu a shoduje se s některými informacemi z báze informací v predikátu.

Poté co jsou zjištěny veškeré potencionálně platné informace, můžeme pro všechny identifikované proměnné nastavit jejich možné hodnoty. Pro každou proměnnou a jí odpovídající vstupní RDF trojice jsem schopen z potencionálně platných informací dohledat jakých hodnot může nabývat za pomoci dosazení objektu či subjektu. Zde opět musíme zvážit všechny tři možnosti (proměnná v subjektu, proměnná v objektu a proměnná v subjektu i objektu).

Z nalezených hodnot, kterých mohou proměnné nabývat, se dále sestaví seznam všech možných kombinací. Tím dostaneme také matici všech možných řešení proměnných vstupního dotazu. Nyní už pouze stačí brát jednu kombinaci za druhou a testovat, zda-li po dosazení konkrétních hodnot místo proměnných do vstupních RDF trojic jsou platné. Platností se v tomto případě myslí, že všechny RDF trojice jsou shodné s informacemi z báze informací. Ze všech takových kombinací, které jsou platné, jsme schopni určit hodnotu hledané proměnné vstupního dotazu. Takto nalezených správných hodnot může být i více, což je očekávané a správné chování a knihovna OpenmRDP na tuto událost samozřejmě umí korektně zareagovat.

6.5.3 Příklad vyhodnocení dotazu

Pro lepší pochopení celého algoritmu vyhodnocení dotazu IDENTIFY si uvedeme příklad odpovídající ukázce z kapitoly 4.3. Uvažujme tedy spojení informací z výpisu 4.1 a výpisu 4.2 doplněné o symetrické informace k informacím umístění jako naši bázi informací a dotaz z výpisu 4.3 jako vstupní dotaz.

Objekt typu Query bude po svém sestavení obsahovat množinu následejících pěti RDF trojic.

```
?material loc:umistenV ?mistnost
?material rdf:typ mat:HorlavyMaterial
<urn:uuid:vrtacka> loc:umistenV ?mistnost
<urn:uuid:vrtacka> task:vrtani ?sur
?sur rdf:typ mat:ZeleznyMaterial
```

Z výše uvedených RDF trojic jsme nyní schopni identifikovat proměnné `?material`, `?mistnost` a `?sur`, ze kterých vvtvoříme objekty typu QueryVariable. Možné hodnoty pro každou proměnnou nebudou prozatím v objektu nastaveny a doplní se až v dalších krocích algoritmu. Dále pro výše uvedené RDF trojice hledáme potencionálně platné informace z informační báze. Proces získání těchto informací je popsán v kapitole 6.5.2 a jeho výsledek bude následující. Pro RDF trojici `?material loc:umistenV ?mistnost` jsou odpovídající potencionálně platné informace:

```
<urn:uuid:vrtacka> loc:umistenV <urn:uuid:mistnost1>
<urn:uuid:box1> loc:umistenV <urn:uuid:mistnost1>
<urn:uuid:olej1> loc:umistenV <urn:uuid:box1>
<urn:uuid:olej1> loc:umistenV <urn:uuid:mistnost1>
```

Pro další RDF trojici, přesněji `?material rdf:typ mat:HorlavyMaterial` je odpovídající potencionálně platná informace pouze jedna.

?material	?mistnost	?povrch
<urn:uuid:vrtacka>	<urn:uuid:mistnost1>	<urn:uuid:povrch1>
<urn:uuid:box1>	<urn:uuid:box1>	
<urn:uuid:olej1>		

Tabulka 6.1: Možné hodnoty proměnných

?material	?mistnost	?povrch
<urn:uuid:olej1>	<urn:uuid:mistnost1>	<urn:uuid:povrch1>
<urn:uuid:box1>	<urn:uuid:box1>	<urn:uuid:povrch1>
<urn:uuid:vrtacka>	<urn:uuid:mistnost1>	<urn:uuid:povrch1>
<urn:uuid:olej1>	<urn:uuid:box1>	<urn:uuid:povrch1>
<urn:uuid:box1>	<urn:uuid:mistnost1>	<urn:uuid:povrch1>
<urn:uuid:vrtacka>	<urn:uuid:box1>	<urn:uuid:povrch1>

Tabulka 6.2: Rozgenerované možnosti hodnot proměnných

```
<urn:uuid:olej1> rdf:typ mat:HorlavyMaterial
```

Třetí RDF trojice `<urn:uuid:vrtacka> loc:umistenV ?mistnost` má také pouze jednu odpovídající potencionálně platnou informaci.

```
<urn:uuid:vrtacka> loc:umistenV <urn:uuid:mistnost1>
```

Čtvrtá trojice ze vstupního dotazu `<urn:uuid:vrtacka> task:vrtani ?sur` odpovídá pro náš příklad také pouze jedné potencionálně platné informaci.

```
<urn:uuid:vrtacka> task:vrtani <urn:uuid:povrch1>
```

A nakonec poslední zmíněná RDF trojice `<urn:uuid:olej1> loc:umistenV <urn:-uuid:mistnost1>` odpovídá následující potencionálně platné informaci.

```
<urn:uuid:povrch1> rdf:typ mat:ZeleznyMaterial
```

Pokud nyní subjekty a objekty potencionálně platných informací zkusíme dosadit za proměnné v původním vstupním dotazu, dostaneme tabulku 6.1. Ze které následným rozgenerováním dostaneme tabulku 6.2 obsahující všechny možné kombinace, které mohou pro dané proměnné a jejich hodnoty nastat.

Z takto vygenerovaných kombinací, z tabulky 6.2, už pouze řádek po řádku bereme jednotlivé hodnoty proměnných, které zkusíme dosadit do vstupního dotazu a pro každou RDF trojici se ptáme instance třídy `InfoManager`, jestli je trojice platná. Narazíme-li na kombinaci, která je platná pro všechny RDF trojice ze vstupního dotazu, považujeme ji za správnou. V našem konkrétním případě je zde pouze jedna platná kombinace a to první řádek z tabulky kombinací 6.2.

Po dosazení hodnot za proměnné do vstupního dotazu bude tento dotaz vypadat následovně.

```
IDENTIFY <urn:uuid:olej1> WHERE
                <urn:uuid:olej1> loc:umistenV <urn:uuid:mistnost1>
                <urn:uuid:olej1> rdf:typ mat:HorlavyMaterial
                <urn:uuid:vrtacka> loc:umistenV <urn:uuid:mistnost1>
                <urn:uuid:vrtacka> task:vrtani <urn:uuid:povrch1>
                <urn:uuid:povrch1> rdf:typ mat:ZeleznyMaterial
```

Je tedy zřejmé že všechny RDF trojice jsou dle báze informací platné a identifikovaná proměnná ?material se rovná hodnotě <urn:uuid:olej1>.

6.6 Veřejná rozhraní

Knihovna OpenMRDP je naimplementována pro použití jak na straně klienta, tak na straně serveru, a proto také má dvě odlišná veřejná rozhraní. Jedním z nich je `OpenmRDPClientAPI` určené pro užití na klientské stanici. Druhým rozhraním je potom `OpenmRDPServerAPI` pro užití na serverech. Programátor, implementující veřejná rozhraní, ke své práci využívá i několik dalších veřejných podpůrných tříd, které jsou blíže popsány v kapitole 6.9. Tyto třídy ovšem využívá pouze k inicializaci veřejných rozhraní a nejsou později zapotřebí. Samozřemostí je, že jsou tyto třídy neměnitelné, a tak nemůže být dodatečně ovlivněno již nainicializované rozhraní.

6.6.1 Rozhraní OpenmRDPClientAPI

Jednodušším z rozhraní je rozhraní `OpenmRDPClientAPI`. Programátor pomocí něj může přistupovat ke dvěma základním funkcionalitám na straně klienta. První funkcionalitou je dříve již několikrát zmiňovaná funkce LOCATE. Metoda zajišťující nalezení lokace zdroje se nazývá `locateResource`. Tato metoda požaduje jako svůj vstupní parametr pouze řetězec, obsahující jméno hledaného zdroje a při úspěšném nalezení vrací jeho URL adresu. V případě, že zdroj nenalezne, vrací prázdný řetězec. Ihned po svém zavolání tato metoda vytvoří zprávu typu LOCATE za pomoci již představené MessageFactory a odešle ji do sítě. Odeslání zprávy probíhá za pomoci protokolu UPD na adresu multicastové skupiny `224.0.24.1` a port 27773. Po následujících 10 vteřin klient aktivně čeká, zda-li nedostane odpověď od některého ze serverů. Nestane-li se tak, klient ukončí čekání se zprávou, že hledaný zdroj nebyl nalezen. V případě, že pro něj některý ze serverů, který naslouchal na multicastové adrese `224.0.24.1`, má nějakou informaci, obdrží klient zprávu typu `Connection Information` 6.3.6, která mu byla doručena s využitím protokolu TCP přímo na jeho konkrétní IP adresu. Ze zprávy Connection Information klient zjistí, zda server vyžaduje autorizaci uživatele a na základě této informace pak naváže se serverem odpovídající spojení (HTTP pro nezabezpečenou nebo HTTPS pro zabezpečenou komunikaci) a odešle HTTP (popřípadě HTTPS) požadavek typu GET. Server mu potom v rámci odpovědi na požadavek zašle odpověď na původní dotaz.

Druhou funkcionalitou je potom možnost identifikace zdroje (IDENTIFY) přístupnou prostřednictvím metody `identifyResource`. Jako vstupní parametr bere metoda pouze řetězec obsahující dotaz a stejného typu je poté i typ návratový. Chování, včetně posloupnosti zasílaných a obdržených zpráv, metody je stejné jako u předchozí funkcionality LOCATE.

Obě veřejné metody jsou ještě přetíženy metodami, které mají navíc parametry pro uživatelské jméno a heslo, sloužící k autorizaci klienta vůči serveru. Tyto metody by měl uživatel volat v případě, že chce se servery komunikovat zabezpečeně, včetně ověření své vlastní identity.

Instanciace třídy `OpenMRDPClientApiImpl` probíhá za pomoci konstruktoru, který jako své argumenty požaduje návratovou adresu, na kterou má později server odesílat odpovědi a instanci třídy `MrdpLogger` 6.10. Pro testovací účely existuje ještě durhý konstruktor, který navíc požaduje boolean příznak, rozhodující o tom, zda má instance běžet v testovacím režimu. Tato třída se také stará o korektní generování sekvenčních čísel jednotlivých zpráv komunikace.

6.6.2 Rozhraní OpenmRDPServerAPI

OpenmRDPServerAPI je dalším veřejným rozhraním knihovny OpenmRDP poskytujícím programátorovi funkcionalitu na serverové straně. První z jeho metod je metoda s názvem `receiveMessages` sloužící k účelu přijímání zpráv. Zde dochází k aktivnímu čekání a měla by se tedy volat v samostatném, server neblokujícím, vlákně. Ihned po přijetí zprávy, reprezentované objektem typu BaseMessage, je na základě operačního řádku rozhodnuto o jaký požadavek se jedná (zda LOCATE či IDENTIFY) a je předáno řízení odpovídajícímu procesoru zpráv. Odpovídající procesor vygeneruje odpověď v podobě ReDEL zprávy a uloží si takto připravenou zprávu do cache paměti reprezentované mapou s klíči typu `ClientEntry` 6.9.1 a hodnotou obsahující odpověď na obdržený dotaz. Nyní, když je odpověď úspěšně vygenerována a připravena v cache paměti, vygeneruje server zprávu typu `Connection Information` 6.3.6 a podle své konfigurace jí nastaví příslušná data. Connection information zprávu server odešle clientovi a dále naslouchá na portu 27741 až se klient o data přihlásí pomocí zprávy typu HTTP/HTTPS GET. Podle klíče ClientEntry dohledá pro daného klienta připravená data v cache paměti a odešle mu je v odpovědi na GET požadavek. Příklad záznamů v takto předchystané cache paměti je uveden na obrázku 6.3

Obrázek 6.3: Ukázka cache paměti s předpřipravenou zprávou.

Jako další metodu toto rozhraní poskytuje metodu `addInformationToInformationBase`, která umožnuje programátorovi přidávat informace do informační báze. Samotné přidání probíhá prostřednictvím informačního manažera a propíše se okamžitě na všechna místa s informační bází spojená. Přímý propis je proveden do souboru informationBase.xml. Změna se promítne samozřejmě v informační bázi načtené v informačním manažerovi a také, pokud je přidávána informace o umístění některého ze zdrojů, dojde k aktualizaci stromu lokací.

Analogicky k metodě addInformationToInformationBase je programátorovi k dispozici metoda `removeInformationFromInformationBase`. Tato metoda provede odstranění informace z báze informací, vymaže záznam o informaci ze souboru informationBase.xml a je-li to třeba aktualizuje také strom lokací.

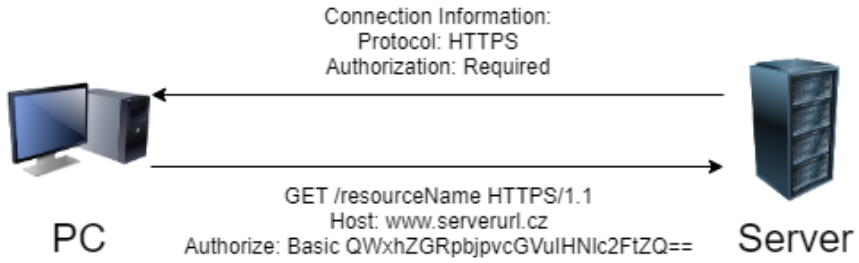
6.7 Bezpečnost

V rámci implementace knihovny OpenmRDP bylo také nutné se zaměřit na bezpečnost, a to jak z hlediska autentizace serverů, tak i autorizace samotných klientů. Pro tento účel jsou

využity prvky standartní komunikace HTTP popřípadě HTTPS. Při instanciaci serverového veřejného rozhraní je vyžadován objekt typu `SecurityConfiguration`, ze kterého si server načte, zda má fungovat v zabezpečeném či nezabezpečeném režimu a vytvoří si podle toho příslušnou instanci třídy pro obsluhu HTTP nebo HTTPS požadavků 6.8. V případě, že má server pracovat v bezpečném režimu, bude pro svou činnost potřebovat platný serverový certifikát, který bude důvěryhodný. V tomto případě bude komunikace probíhat za pomoci protokolu HTTPS a klient si tak pomocí certifikátu může ověřit skutečnou identitu serveru a je tak tedy zajištěna autenticita informací poskytovaných serverem.

Pokud to server vyžaduje, je nutné, aby se uživatel na serveru nejprve autorizoval. K tomuto účelu je zde vystaveno další rozhraní nesoucí název `UserAuthorizator`, které by si měl každý uživatel knihovny openmRDP doplnit o vlastní implementaci. Nutnost vlastní implementace je zde zejména kvůli tomu, že každý server může mít autorizování uživatelů zajišťován jiným způsobem a není tedy možné napsat univerzální autorizátor, který by byl vždy funkční. Uvažme třeba ověření uživatelů vůči databázi a naproti tomu ověření vůči `Active Directory`. Pro účely testování knihovny zde je testovací implementace `UserAuthorizatorTestImpl`, která pouze oveřuje zadané přihlašovací údaje oproti statickému seznamu.

Jak již bylo zmíněno, pro bezpečnou komunikaci se využívá protokol HTTPS. Pokud se tedy uživatel bude ověřovat, je použita technologie `Basic access authentication` 5.2.1, kde jednou z hlaviček HTTPS požadavku je i hlavička `Authorization` nesoucí zakódované přihlašovací údaje. Pro zakódování a dekódování se využívá technologie `Base64`. Příklad takového ověření je uveden na obrázku 6.4



Obrázek 6.4: Ověření klienta za pomocí Basic access authentication.

Pokud se již jednou uživatel přihlásil, je tato informace udržována v paměti serveru a proces autorizace uživatele se již znovu neprovádí. Provede se pouze kontrola hash rětězce a IP adresy, vůči informacím, které si server drží ve své cache paměti. Pokud nepřijde pro konkrétního uživatele požadavek na server v čase delším než půl hodiny, dojde k odstranění záznamu z cache paměti a při dalším požadavku je nutné provést celý proces ověření uživatele znovu.

6.8 Obsluha HTTP požadavků

Poté co server odešle klientovi zprávu Connection Information obsahující adresu pro připojení, je nutné na této adrese naslouchat. K tomuto účelu slouží třídy pro obsluhu požadavků implementující standartní rozhraní `com.sun.net.httpserver.HttpHandler`, které deklaruje jednu metodu s názvem `handle`. Pokud tedy přijde nějaký požadavek na server, je automaticky zavolána tato metoda, kde k obsluze dochází. V závisloti od toho jaké informace byly při instanciaci serverového rozhraní openmRDPServerAPIImpl 6.6.2 předány

prostřednictvím objektu ServerConfiguration 6.9.3 (přesněji jeho vnořeného objektu SecurityConfiguration 6.9.2), je vytvořena buď instance třídy `NonSecureServerHandler` 6.8.1 nebo `SecureServerHandler` 6.8.2.

6.8.1 NonSecureServerHandler

Pro běh serveru, který nevyužívá zabezpečenou komunikaci, a tedy ani nepodporuje ověřování uživatelů, se využívá třída `NonSecureServerHandler`, která se nachází v balíku server. Při instanciaci této třídy se jako parametr konstruktoru požaduje mapa sloužící jako cache paměť pro připravené zprávy. Klíčem této mapy je objekt typu `ClientEntry` 6.9.1 a hodnotou potom samotná zpráva reprezentována objektem typu `BaseMessage` 6.3.

Celá funkcionalita metody handle tedy spočívá pouze v tom, že je z příchozí zprávy získána IP adresa a sekvenční číslo zprávy. Pomocí těchto údajů je sestaven odpovídající objekt typu ClienEntry a nalezena zpráva v mapě obdržené při vytváření instance třídy. S takto připravenou zprávou sestavíme odpovídající odpověď, kterou odešleme zpátky klientovi a odstraníme zprávu z mapy připravených zpráv.

6.8.2 SecureServerHandler

V případě, že server disponuje certifikátem může probíhat komunikace zabezpečeně. O tom je rozhodnuto prostřednictvím informací při vytváření nové instance implementace rozhraní `OpenmRDPServerAPI` 6.6.2.

Narozdíl od nezabezpečené obsluhy požadavků, požaduje konstruktor, pro instanciaci třídy obsluhy požadavků zabezpečené komunikace, parametrů více. Jedním z nich je samozřejmě, stejně jako v předchozím případě, mapa sloužící k uchování připravených zpráv pro klienty. Dále pak objekt typu `UserAuthorizator` 6.9.4, který bude sloužit k ověřování identity klienta a mapa pro udržení informací o již ověřených uživatelích.

Poslední zmíněná mapa pro udržení informací o již zmíněných uživatelích, má jako klíč objekt typu `ClientAccessMetadata` 6.9.5 a jako hodnotu potom login ověřeného klienta. Platnost záznamu v této mapě je půl hodiny a po jejím uplynutí se klient musí znovu autorizovat.

Při příchozím požadavku se obdobně jako při nezabezpečené komunikaci z HTTPS požadavku získá IP adresa klienta a sekvenční číslo zprávy v komunikaci. Navíc je zde přítomna hlavička `Authorization`, ve které se nachází `Base64` hash. Obsah hlavičky `Authorization` je dekódován pomocí standartní třídy jazyka Java `java.util.Base64`. Z dekódovaného řetězce potom získáme login a heslo klienta, kterého můžeme pomocí instance třídy `UserAuthorizator` ověřit. Nastane-li případ, že uživatel ověřen není, je klientovi odeslána zpět odpověď se stavovým kódem `403 - FORBIDDEN` a zprávou `"Invalid login or password."`. Jsou-li však přihlašovací údaje v pořádku a uživatel je ověřen, uložíme o tom příslušné informace do mapy ověřených uživatelů, sestavíme odpovídající objekt typu `ClientEntry` a z mapy předpřipravených odpovědí vybereme odpovídající odpověď. Ta je následně odeslána v odpovědi na HTTPS dotaz a vymazána z mapy předpřipravených odpovědí.

6.9 Podpůrné objekty

Knihovna ve své implementaci využívá několik podpůrných objektů, které pomáhají udržovat data. Jedná se například o některé doménové objekty či takzvané Data Transfer Objekty. Jednotlivé podpůrné objekty jsou blíže popsány v následujících podkapitolách.

6.9.1 ClientEntry

Objekt `ClientEntry` je využíván jako klíč pro mapu sloužící jako cache paměť, ve které se uchovává již připravená odpověď pro klienta. Tento objekt se skládá z následujících atributů:

- **address** – adresa klienta, pro kterého je zpráva určena
- **sequenceNumber** – sekvenční číslo zprávy, která nesla odpovídající dotaz pro již vygenerovanou odpověď
- **created** – časový údaj o vytvoření záznamu

6.9.2 SecurityConfiguration

`SecurityConfiguration` v sobě uchovává informace, které jsou důležité pro chod serveru. Je zde nastavení určující zda server podporuje zabazpečené spojení HTTPS a zda bude vyžadována po uživateli autorizace. Dále se zde nachází samotný autorizátor uživatelů reprezentován pomocí rozhraní `UserAuthorizator`, který server využívá k ověření identity uživatele a informace k serverovým certifikátům.

6.9.3 ServerConfiguration

Tento objekt nese pouze základní informace o konfiguraci serveru, a to konkrétně jeho IP adresu, port na kterém bude probíhat HTTP/HTTPS komunikace a vnořený objekt nastavení zabezpečení, který je instancí třídy `SecurityConfiguration` 6.9.2. ServerConfiguration objekt se používá při vytváření nové instance třídy OpenmRDPServerAPIImpl 6.6.2 a server z něj tyto uložené informace čerpá pro své chování.

6.9.4 UserAuthorizator

Jedním z hlavních objektů z hlediska dosažení bezpečné komunikace je třída `UserAuthorizator`. Vzhledem k tomu, že může existovat opravdu velké množství způsobů jak ověřit uživatele, jedná se ve skutečnosti pouze o rozhraní, definující jak by měla výsledná implementace vypadat. Toto rozhraní disponuje jednou jedinou metodou s názvem `authorizeUser`, která vrátí hodnotu datového typu boolean.

Výsledná implementace této třídy je ponechána na programátorovi, který se rozhodne knihovnu OpenmRDP využít. Pro účely testování je zde pouze jedna implementace, a to třída `UserAuthorizatorTestImpl`. V předpřipraveném statickém seznamu známých a povolených uživatelů je uživatel s přihlašovacím jménem "testUser"a heslem "Password123". Využití této implementace v produkčním prostředí se silně nedoporučuje a dokonce může působit jako bezpečnostní riziko.

6.9.5 ClientAccessMetadata

Objekt typu `ClientAccessMetadata` je využíván jako klíč pro mapu již ověřených klientů. Pomáhá optimalizovat proces přihlášení tak, aby nebylo nutné ověřovat klienta za pomoci třídy `UserAuthorizator` 6.9.4 při každém požadavku od klienta. Atributy tohoto objektu jsou následující:

- **clientAddress** – adresa klienta, který je již autorizován, ve formě řetězce

- **lastAccess** – objekt datového typu `Instant`, který uchovává časový údaj o posledním přístupu klienta k serveru

6.10 Zaznamenávání událostí knihovny

Pokud programátor chce jakkoliv monitorovat činnost knihovny OpenmRDP, zejména pak v případech, kdy nastává nějaká z očekávaných vyjímek, má k dispozici zaznamenávání zpráv (událostí) za pomocí rozhraní `MrdpLogger`. Toto rozhraní definuje tři základní metody, které se na některých místech v implementaci knihovny využívají.

- **logDebug** – Využívá se na místech, kde dochází k časově kritickým operacím. Například při vyhodnocování dotazů.
- **logInfo** – Poskytuje obecné zaznamenávání událostí. Například pokud dorazí na server nějaký mRDP požadavek.
- **logError** – Slouží k zaznamenávání chybových hlášek, které mohou při běžném provozu nastat jako například chybný formát příchozí zprávy.

Ve vlastní implementaci knihovny jsou k dispozici dvě standartní implementace rozhraní `MrdpLogger`. První je testovací implementace `MrdpTestLoggerImpl`, která vypisuje všechny tři stupně zpráv na standartní výstup. Druhou implementací je pak `MrdpDummyLogger`, který nevykonává vůbec žádnou práci. Veškeré informace, které mají být zaznamenány jsou ztraceny. Jeho využití je plánováno pro případy, kdy programátor nechce nikde uchovávat informace o chodu knihovny.

Kapitola 7

Testování knihovny

V průběhu samotné implementace knihovny OpenmRDP byl kladen velký důraz i na testování již implementovaných částí. Naprostá většina funkcionality je pokryta automatizovanými *Unit* testy. Pro implementaci těchto testů byla použita knihovna třetí strany, konkrétně JUnit ve verzi 4.12. Samozřejmostí je potom i implementace vlastního serveru poskytujícího funkcionalitu knihovny OpenmRDP a klientů k němu přistupujícím. Jeden z klientů je aplikace určená pro klasické stolní počítače s ovládáním pomocí příkazové řádky, druhým pak android aplikace určena pro mobilní telefony s operačním systémem android. Implementace aplikace serveru a klientů bude detailněji popsána v následujících podkapitolách.

7.1 Automatizované testy

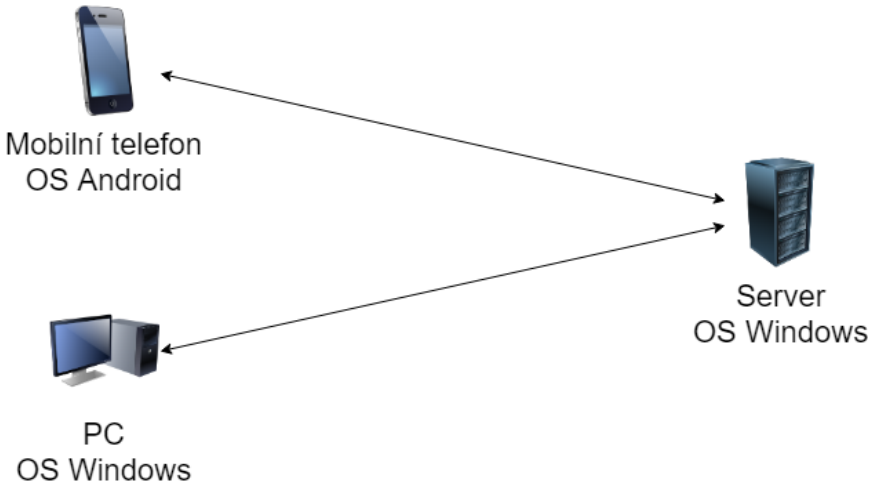
Každý logický celek aplikace obsahuje i své pokrytí plně automatizovanými testy. V případě, že některá z částí nebyla možná jednoduše pokrýt testy, například z důvodu závislosti na externích souborech nebo nutnosti komunikace s dalšími mechanismy (ověření identity klienta), byla tato závislost nahrazena rozhraním a testovací implementací. Příklady těchto implementací jsou například následující:

- **UserAuthorizatorTestImpl** – Nahrazuje produkční implementaci rozhraní `UserAuthorizator`. Ověří předem definovanou množinu uživatelů.
- **MessageReceiverTestImpl** – Generuje náhodné příchozí zprávy namísto jejich přijímání ze sítě.
- **MessageSenderTestImpl** – Namísto toho, aby byla zpráva odeslána do sítě, je pouze vypsána na standartní výstup.
- **MrdpTestLoggerImpl** – V místech, kde je vyžadováno logování, je použita testovací implementace loggeru, který pouze vypisuje zprávy na standartní výstup.
- **OntologyTestService** – Slouží jako náhrada za produkční třídu pro načítání ontologie. Tato implementace načítá ontologii ze statického seznamu.

Všechny testy jsou k dispozici v adresáři `\openmRDP\test` a mohou být kdykoliv opakovaně spouštěny.

7.2 Testovací prostředí

Pro účel vývoje a testování vzniklo i testovací prostředí skládající se z mobilního telefonu a dvou pracovních stanic, z nichž jedna plní funkci serveru a druhá klienta. Výsledná topologie testovacího prostředí je zobrazena na obrázku 7.1.



Obrázek 7.1: Testovací topologie knihovny openmRDP.

7.2.1 Testovací server

Testovací server byl naprogramován v programovacím jazyce Java ve verzi 1.8 s využitím knihovny OpenmRDP. Jedná se o jednoduchou konzolovou aplikaci, která ihned po svém spuštění využívá implementace knihovny. Server není nijak platformně omezen a k jeho běhu stačí pouze běhové prostředí jazyku Java. Příklad takového serveru je uveden v ukázce 7.1.

```
UserAuthorizator userAuthorizator = new UserAuthorizatorTestImpl();

SecurityConfiguration securityConfiguration =
       SecurityConfigurationFactory.
              createSecureSecurityConfiguration(userAuthorizator);

ServerConfiguration serverConfiguration =
       new ServerConfiguration(SERVER_IP_ADDRESS, MRDP_PORT);

MrdpLogger logger = new MrdpTestLoggerImpl();

OpenmRDPServerAPI api =
       new OpenmRDPServerAPIImpl(
              securityConfiguration,
              serverConfiguration,
              logger
       );

try {
       api.receiveMessages();
} catch (AddressSyntaxException | NetworkCommunicationException e) {
       logger.logError(e.getMessage());
}
```

Výpis 7.1: Implementace serveru s podporou knihovny OpenmRDP

7.2.2 Klient pro stolní počítač

Stejně jako testovací server je i klientská aplikace se jmeném `OpenmRDPTestClient` naprogramována v programovacím jazyce Java ve verzi 1.8. Stačí pouze mezi importy zahrnout implementaci knihovny OpenmRDP a využívat její klientské veřejné rozhraní. Aplikace je multiplatformní a k jejímu běhu stejně jako v případě testovacího serveru stačí pouze běhové prostředí jazyka Java. Aplikace je konzolová a při svém spuštění umožňuje několik následujících přepínačů.

- **-l** – určuje, že klientská aplikace bude posílat požadavek typu `LOCATE` a je následován jménem zdroje, který chce uživatel lokalizovat.
- **-i** – udává, že klientská aplikace bude odesílat požadavek typu `IDENTIFY`. Tento přepínač je následován dotazem, který chce klient odeslat na server.
- **-if** – poskytuje stejnou funkcionalitu jako přepínač `-i` s rozdílem, že odesílaný dotaz bude uložen v souboru, jehož cesta je udána za přepínačem.
- **-e** – udává koncovou část adresy (takzvaný "endpoint") na který bude server odpovídat.
- **-u** – za tímto přepínačem následuje uživatelské jméno uživatele.
- **-p** – slouží pro zadání hesla pro příhlášení uživatele.

Přepínače `-l`, `-i` a `-if` jsou výlučné a jejich kombinace není povolena. Validní příkazy pro spuštění klientské testovací aplikace jsou tedy například následující:

```
OpenmRDPTestClient -l urn:uuid:olej1
OpenmRDPTestClient -l urn:uuid:olej1 -e testEndpoit
OpenmRDPTestClient -l urn:uuid:olej1 -u testUser -p Password123
```

Naopak nevalidní kombinace přepínačů by potom vypadala následovně:

```
OpenmRDPTestClient -l urn:uuid:olej1
       -i ?material WHERE ?matiral <loc:locatedIn> urn:uuid:mistnost1
```

Zakomponování knihovny OpenmRDP do vlastní klientské aplikace určené pro stolní počítače, je stějně jednoduché jako v případě implementace testovacího serveru. Krátký příklad je uveden v ukázce 7.2.

```
MrdpLogger logger = new MrdpTestLoggerImpl();
OpenmRDPClientAPI api = new OpenmRDPClientApiImpl(TEST_ENDPOINT, logger);

try {
       api.locateResource("urn:uuid:olej1");
} catch (NetworkCommunicationException e) {
       logger.logError(e.getMessage());
}
```

Výpis 7.2: Implementace klientské aplikace pro stolní počítač s podporou knihovny OpenmRDP

Odpověď, kterou nám testovací server implementující knihovnu OpenmRDP vrátí, bude po jejím obdržení v testovací klientské aplikaci vypsána na standartní výstup.

7.2.3 Aplikace pro operační systém Android

Poslední testovací aplikací je aplikace pro operační systém android pojmenová jako `openMRDPDemoApp`. Minimální verze operačního systému Android je stanovena na verzi `4.0.3 ICE_CREAM_SANDWICH` a to jak pro mobilní telefon tak tablet. Tato jednoduchá aplikace se skládá z pouhých 5 aktivit, které jsou postačující pro plnou demonstraci funkcionality knihovny a jsou následující:

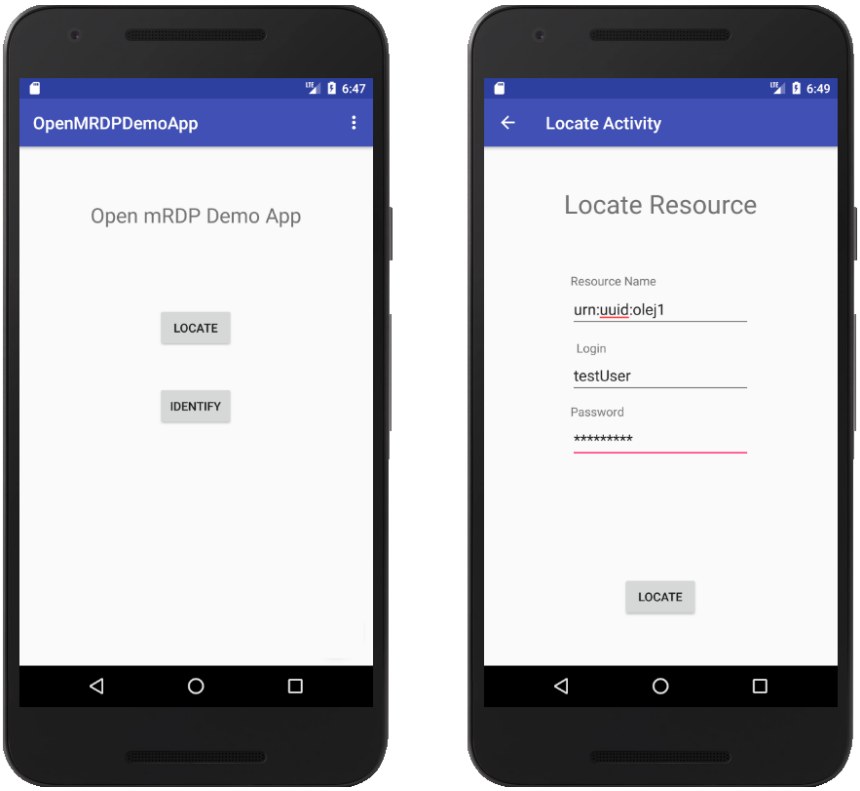
- **activity_main** – Úvodní aktivita aplikace, která obsahuje základní menu s výběrem operací. Ukázka je zobrazena na obrázku 7.2.
- **identify_activity** – Aktivita umožňující zadat uživateli dotaz na zdroj a popřípadě zadat přihlašovací údaje.
- **identify_result_activity** – Slouží pro zobrazení výsledku z dotazu `IDENTIFY`.
- **locate_activity** – Aktivita má stejný design jako aktivita `identify_activity` pouze slouží k odesílání dotazů typu `LOCATE`. Ukázka je společně s aktivitou `activity_main` k nahlédnutí na obrázku 7.2.
- **locate_result_activity** – Zobrazuje výsledek dotazu typu `LOCATE`.

Pro své spuštění mobilní aplikace využívá následující dvě oprávnění, které musí uživatel před instalací aplikaci povolit:

- **android.permission.INTERNET** – umožňuje aplikaci otevírat síťové sockety a komunikovat prostřednictvím sítě
- **android.permission.ACCESS_NETWOR_STATE** – umožňuje aplikaci přístup k informacím o dostupných sítích

Android aplikace jsou oproti klasickým aplikacím pro stolní počítače odlišné v přístupu k síti. Nedovolují nám přistupovat k síti v hlavní aktivitě aplikace z důvodu pomalejší odezvy aplikace, a proto musí komunikace prostřednictvím knihovny OpenmRDP se serverem probíhat v samostatném asynchronním vlákně. K tomuto účelu slouží třída `OpenMRDPApiTask`, která dědí od třídy `AsyncTask` a překrývá její metodu `DoInBackground`. V této metodě je voláno klientské veřejné rozhraní knihovny OpenmRDP 6.6.1 a vrací do aktivit, které tento asynchronní úkol spustily, výsledek ve formě řetězce.

Jediné aktivity, které spouští tento asynchronní úkol jsou aktivity `LocateActivity` a `IdentifyActivity` a po jeho dokončení předávají výsledek do aktivit zobrazujících výsledek.



Obrázek 7.2: Ukázka mobilní testovací aplikace.

Pro účely testování je tato testovací android aplikace doplněna také o vlastní implemantaci rozhraní `MrdpLogger` 6.10, která je pojmenována jako `MrdpAndroidLogger` a pro logování využívá standartního nástroje pro vytváření logů v operačním systému android `android.util.Log`.

Kapitola 8

Využití knihovny OpenmRDP v praxi

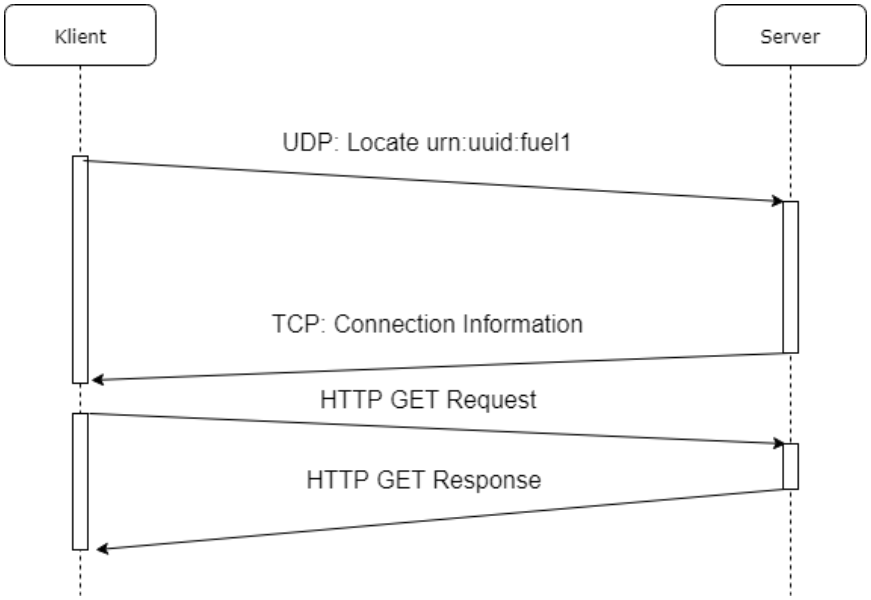
Knihovna OpenmRDP má v praxi daleko širší využití, než příklad popsaný v podkapitole 4.3. Do budoucna by se dala použít například k nalezení lokace nejbližších dostupných zdrojů v komplexu budov nebo určování polohy předmětu, který se volně pohybuje v prostoru. Uvažme třeba dvě budovy, které obsahují několik pater a množství místností. Pokud vhodně zkonstruujeme dotaz a server bude mít dostatečné informace o umístění zdroje, může nás ihned nasměrovat k nejbližšímu zdroji, což se dá využít například pro síťové tiskárny. Druhý zmiňovaný příklad může být třeba zdroj, který má přístup k serverovému API a při každé změně lokace o sobě změní informaci v informační bázi. Jeho nalezení je potom pro klienta otázkou jednoho dotazu na server a ihned ví, kde se hledaný zdroj nachází.

8.1 Síťový provoz při využití knihovny OpenmRDP

Pro demonstraci síťové komunikace využijeme testovací příklad uvedený v podkapitole 4.3. Na straně klienta se nachází pracovní stanice s operačním systémem `Ubuntu 14.04` a klientská testovací aplikace představená v podkapitole 7.2.2. Na druhé straně, tedy na straně serveru, je poté pracovní stanice s operačním systémem `Windows 10` a serverovou aplikací představenou v podkapitole 7.2.1. Klientská aplikace během demonstrace vyhledává informace o umístění zdroje `urn:uuid:fuel1` a obdrží odpověď od nezabezpečeného serveru. Sekvenční diagram této komunikace je znázorněn na obrázku 8.1.

Nejdříve zašle klient směrem k serverům na dřívě specifikovanou multicastovou skupinu prostřednictvím protokolu UDP zprávu typu mRDP LOCATE. Pokud některý ze serverů má pro klienta odpověď, odešle mu prostřednictvím protokolu TCP zpět zprávu typu `Connection Information`. Jak odeslaný požadavek, tak odpověď od serveru je možno vidět na obrázku v příloze A.1

Nyní když klient ví, že pro něj konkrétní server má připravenou požadovanou odpověď na původní dotaz, naváže se serverem HTTP spojení na základě informací, které obdržel v předchozí zprávě. V odpovědi na předchozí HTTP GET požadavek od klienta odešle server HTTP odpověď s připravenou odpovědí na původní dotaz. Celá tato HTTP komunikace je uvedena na obrázku v příloze B.1



Obrázek 8.1: Sekvenční diagram komunikace klient server.

8.2 Dostupnost knihovny a aplikací

Knihovna byla programována v programovacím jazyce Java 1.8 a je k dispozici pod licencí `open-source`, volně ke stažení v repozitáři na serveru `https://github.com/socker-01/openmRDP`.

Testovací klientská aplikace 7.2.2, testovací serverová aplikace 7.2.1 ani android aplikace 7.2.3 pod licencí `open-source` již nejsou, nicméně jsou taktéž k nahlédnutí v následujících repozitářích.

- **Klientská aplikace** – `https://github.com/socker01/openmRDPTestClient`
- **Serverová aplikace** – `https://github.com/socker01/openmRDPTestServer`
- **Android aplikace** – `https://github.com/socker01/openMRDPDemoApp`

Kapitola 9

Závěr

Diplomová práce se zabývá implementací knihovny pro bezpečné nalezení zdrojů REST architektury na základě jejich sémantického popisu. Toto téma jsem si zvolil zejména z důvodu, že jejím výstupem má být open-source knihovna, která může být později využívána odbornou veřejností. Témeř celý teoretický návrh této práce vycházel z návrhu protokolu mRDP. Po emailové komunikaci s jedním z autorů tohoto protokolu (konkrétně s panem doktorem Vazquezem) jsem se dozvěděl, že nikdy nebyla realizována implementace. Od pana doktora jsem také získal svolení k využití znalostí, které v návrhu protokolu byly uvedeny.

Pro účely návrhu a analýzy řešení do velké míry stačil teoretický návrh zmiňovaného protokolu, nastudování znalostí z oblasti protokolu HTTP a osvojení znalosti mechanismu sémantického vyhledávání. Velká část nastudovaných znalostí je již popsána v kapitolách této diplomové práce.

V začátcích realizace diplomové práce bylo připraveno adekvátní vývojové a testovací prostředí tak, jak bylo popsáno v druhé kapitole. Před samotnou implementací knihovny byly všechny návrhy řešení prokonzultovány s vedoucím práce.

V průběhu implementace knihovny jsem narazil na několik problémů, které mě vedly k několika drobným odchylkám od původního teoretického návrhu. První z těchto odchylek je posloupnost komunikace mezi klientem a serverem, která musela být rozšířena o jeden nový typ zprávy (Connection Information), a to především z důvodu obtížného zakomponování knihovny do operačního systému Android. Druhou poté změna navrženého portu pro mRDP komunikaci z portu 2773, který byl již bohužel rezervovaný, na port 27773. Ve všech ostatních bodech se implementace shoduje s původním teoretickým návrhnem.

Knihovna je v současné době připravena k využití v produkčních aplikacích a její implementace je velmi jednoduchá, což je patrné například i z uvedených příkladů v kapitole 7, zabývající se testováním knihovny. Všechny testovací aplikace, včetně knihovny samotné, jsou k dispozici v online repozitářích na serveru `GitHub`.

Části knihovny, které jsou veřejně dostupné programátorovi, jsou podrobně zdokumentovány prostřednictvím nástroje `JavaDoc`, tak aby bylo jejich využití co nejsnadnější.

Do budoucna je zde ještě prostor pro vylepšení síťové komunikace tak, aby odpovídala původnímu teoretickému návrhu. Je zde také určitě prostor pro optimalizaci algoritmu řešení vstupních dotazů a pro celkové rozšíření funkcionality knihovny.

Literatura

[1] AHMAD, R. M. C. a. M. S. A., Mohammad Nazir: *Ontology-Based Applications for Enterprise Systems and Knowledge Management*. IGI Global, 2012, ISBN 9781466619937.

[2] BRADLEY, N.: *XML Companion, The, Third Edition*. Addison-Wesley Professional, 2001, ISBN 9780201770599.

[3] CARLSON, J.: *REST vs. SOAP APIs: Worlds Apart*. Červen 2016, [Online; navštíveno 30.12.2017].
URL https://rigor.com/blog/2016/06/restvs-soapapis

[4] CHEN Harry, J. A., FININ Tim: *Dynamic service discovery for mobile computing: Intelligent agents meet Jini in the Aether. Baltzer Science Journal on Cluster Computing*, 2001: str. 343–354.

[5] MOTEJLKOVÁ, A.: *Technologie sémantického webu*. Červen 2011, [Online; navštíveno 03.01.2018].
URL https://ikaros.cz/technologie-semantickeho-webu

[6] PATNI, S.: *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS*. Apress, 2017, ISBN 9781484226650.

[7] PINKEL, M. S. T. H. G. L. C.: *SP2Bench: A SPARQL Performance Benchmark*. [Online; navštíveno 04.01.2018].
URL https://arxiv.org/pdf/0806.4627.pdf

[8] POWERS, S.: *Practical RDF*. O'Reilly Media, Inc., 2003, ISBN 9780596002633.

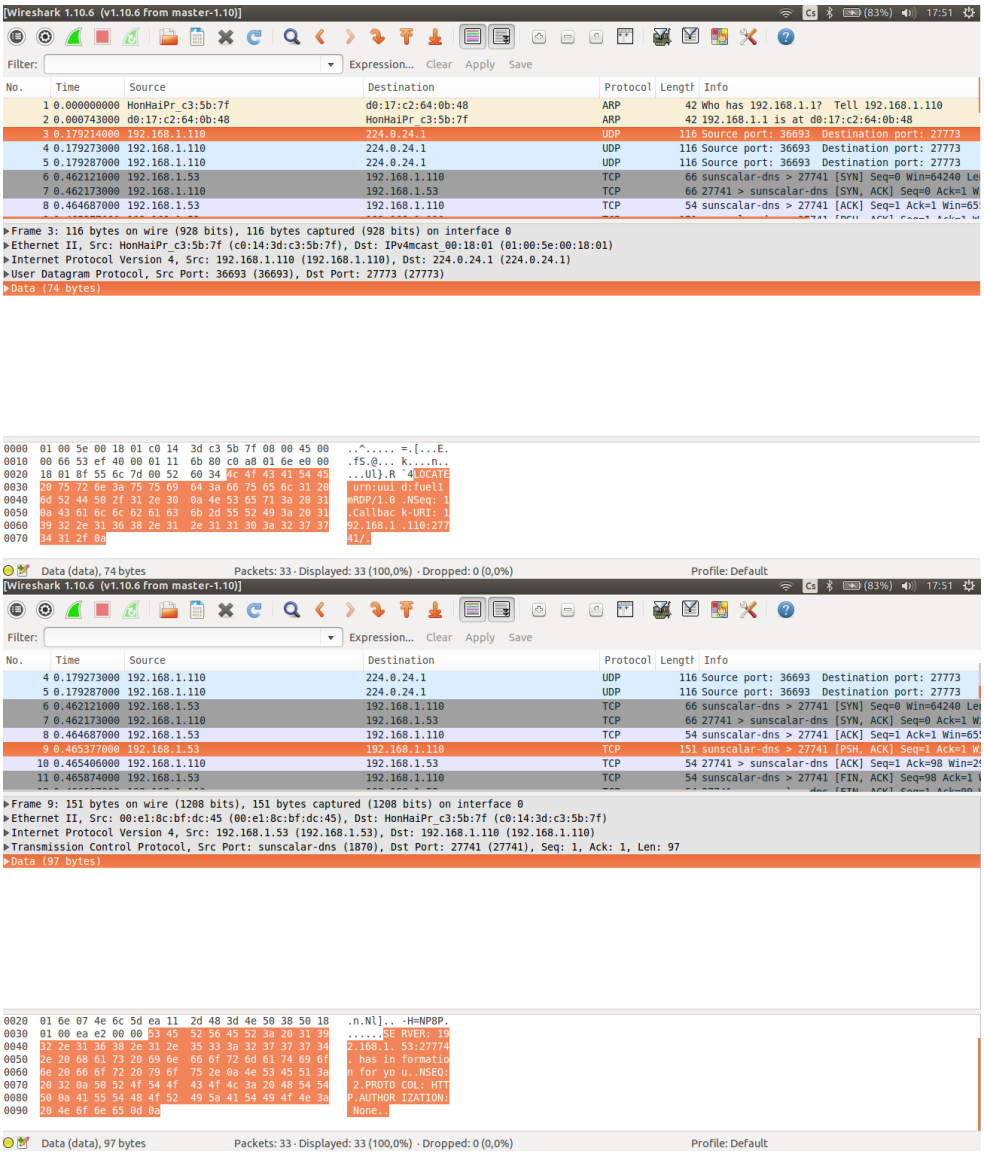
[9] TECHNOLOGIES, J. a. I.: *Network Protocols Handbook*. Javvin Press, 2007, ISBN 9781602670020.

[10] VAZQUEZ, D., J.I.; LÓPEZ-DE-IPINA: *mRDP: An HTTP-based lightweight semantic discovery protocol. Computer Networks*, ročník 51, č. 16, 2007: s. 4529–4542, ISSN 1389-1286.

[11] VAZQUEZ, J.: *A Reactive Behavioural Model for Context-Aware Semantic Devices*. Phd thesis, Universidad de Deusto, 48007 Bilbao, Spain, 2007.

Příloha A

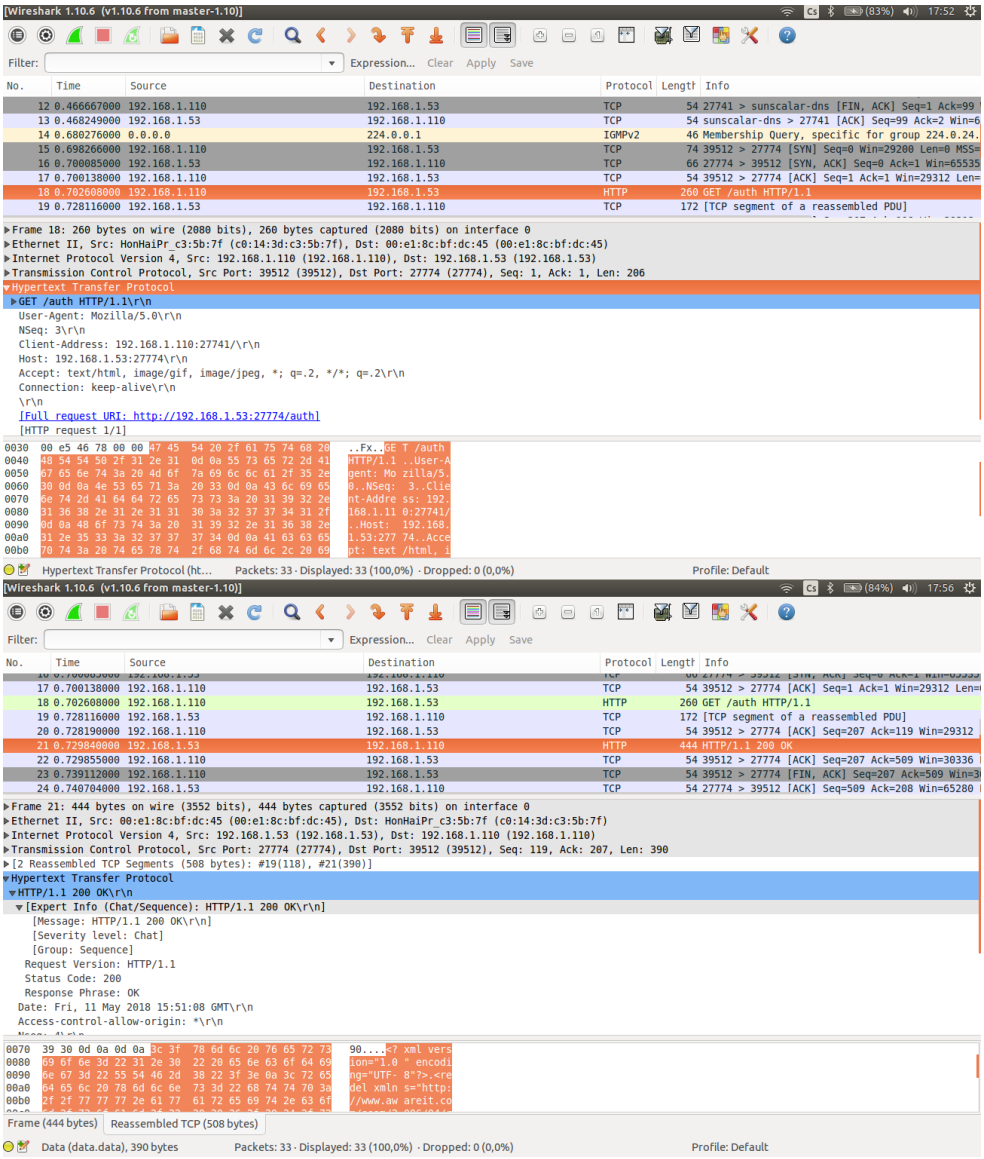
Příklad mRDP dotazu a odpovědi



Obrázek A.1: Příklad mRDP LOCATE zprávy a Connection Information odpovědi.

Příloha B

Příklad HTTP komunikace



Obrázek B.1: Příklad HTTP komunikace mezi klientem a serverem.